QCDNUM: Fast QCD Evolution and Convolution

QCDNUM Version 17.00/06

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

July 10, 2012

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. Other types of evolution can be accessed by feeding alternative sets of evolution kernels into the program. A versatile convolution engine provides tools to compute parton luminosities, cross-sections in hadron-hadron scattering, and deep inelastic structure functions in the zero-mass scheme or in generalised mass schemes. Input to these calculations are either the QCDNUM evolved densities, or those read in from an external parton density repository. Included in the software distribution are 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: 17.00
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** 5  

2 **QCD Evolution** 6  
   2.1 Evolution of the Strong Coupling Constant 6  
   2.2 The DGLAP Evolution Equations 7  
   2.3 Renormalisation Scale Dependence 9  
   2.4 Decomposition into Singlet and Non-singlets 10  
   2.5 Flavour Number Schemes 12  

3 **Numerical Method** 15  
   3.1 Polynomial Spline Interpolation 15  
   3.2 Convolution Integrals 18  
   3.3 DGLAP Evolution 20  

4 **The QCDNUM Program** 24  
   4.1 Source Code 24  
   4.2 Application Program 24  
   4.3 Validation and Performance 27  

5 **Subroutine Calls** 29  
   5.1 Initialisation 31  
   5.2 Grid Definition 32  
   5.3 Weights 35  
   5.4 Parameters 36  
   5.5 Evolution 39  
   5.6 External Pdfs 41  
   5.7 Pdf Interpolation 42  
   5.8 Fast Interpolation 44  

6 **Convolution Engine** 44  
   6.1 Rescaling Variable in Convolution Integrals 45  
   6.2 Weight Tables 46  
   6.3 Convolution 52  
   6.4 Interpolation 55
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 $\mu^2$ characteristic of the hard scattering process. Whereas the $x$-dependence of the parton densities is non-perturbative, the $\mu^2$ 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 $\alpha_s$ [2, 3].

QCDNUM is a FORTRAN program that numerically solves the DGLAP evolution equations on a discrete grid in $x$ and $\mu^2$. Input to the evolution are the $x$-dependence of the parton densities at some input mass factorisation scale, and an input value of $\alpha_s$ 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 2 GHz processor it takes less than 10 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. Alternative sets of evolution kernels can be fed into the program to perform other types of evolution. It is also possible to read a parton density set from an external library, instead of evolving these from the input scale.

A versatile set of convolution routines is provided that can be used to calculate hadron-hadron scattering cross-sections, parton luminosities, or deep inelastic structure functions in either the zero-mass or in generalised mass schemes. Included in the software distribution are the ZMSTF and HQSTF add-on 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. The QCDNUM convolution engine is presented in Section 6. The ZMSTF and HQSTF packages are described in the Appendices C and D, respectively.
2 QCD Evolution

In pQCD, the strong coupling constant \( \alpha_s \) evolves on the renormalisation scale \( \mu_R^2 \). The starting value is specified at some input scale, which usually is taken to be \( m_Z^2 \).

The parton density functions (pdf) evolve on the factorisation scale \( \mu_F^2 \). The starting point of a pdf evolution is given by the \( x \) dependence of the pdf at some initial scale \( \mu_0^2 \).

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 \( n_f \) which specifies how many quark species (d, u, s, \ldots) are participating in the QCD dynamics. In the fixed flavour number scheme (FFNS), \( n_f \) is kept fixed throughout the evolution. Input to an FFNS evolution are then the gluon density and \( 2n_f \) (anti-)quark densities at the input scale \( \mu_0^2 \). In the variable flavour number scheme (VFNS), the flavour thresholds \( \mu_{c,b,t}^2 \) are introduced and 3 light quark densities (d, u, s) are, together with the corresponding anti-quark densities, specified below the charm threshold \( \mu_c^2 \). 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 \( \mu_R^2 \neq \mu_F^2 \). QCDNUM supports a linear relationship between the two scales.

In the following sections we describe the evolution of \( \alpha_s \) 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,

\[
\frac{d\alpha_s(\mu^2)}{d \ln \mu^2} = -\sum_{i=0}^{2} \beta_i a_s^{i+2}(\mu^2). \tag{2.1}
\]

Here \( \mu^2 = \mu_R^2 \) is the renormalisation scale and \( a_s = \alpha_s/2\pi \). The \( \beta \)-functions in (2.1) depend on the number \( n_f \) of active quarks with pole mass \( m < \mu \). In the \( \overline{\text{MS}} \) scheme they are given by [9, 10]

\[
\begin{align*}
\beta_0 &= \frac{11}{2} - \frac{1}{3} n_f \\
\beta_1 &= \frac{51}{2} - \frac{19}{6} n_f \\
\beta_2 &= \frac{2857}{16} - \frac{5033}{144} n_f + \frac{325}{432} n_f^2.
\end{align*} \tag{2.2}
\]

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

\[
\frac{1}{a_s(\mu^2)} = \frac{1}{a_s(\mu_0^2)} + \beta_0 \ln \left( \frac{\mu^2}{\mu_0^2} \right) \equiv \beta_0 \ln \left( \frac{\mu^2}{\Lambda^2} \right). \tag{2.3}
\]
In (2.3), the parameter \( \Lambda \) is defined as the scale where the first term on the right-hand side vanishes, that is, the scale where \( \alpha_s \) 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 \( \alpha_s \) at any other scale is then obtained from a numerical integration of (2.1),\(^1\) instead of from approximate analytical solutions parameterised in terms of \( \Lambda \).

In the evolution of \( \alpha_s \), 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 \( \alpha_s \) evolution at the flavour thresholds [11]; see Section 2.5 for details.

In Figure 1, we plot the evolution of \( \alpha_s \) calculated at LO, NLO and NNLO.\(^2\) Because pQCD breaks down when \( \alpha_s \) 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 \( \mu^2 \) at larger perturbative order.

2.2 The DGLAP Evolution Equations

The DGLAP evolution equations can be written as

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

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.

---

\(^1\)I thank A. Vogt for providing his 4th order Runge-Kutta routine to integrate (2.1) up to NNLO.

\(^2\)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 \( \mu^2 \). 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 \( f(x, \mu^2)dx \) is, at a given \( \mu^2 \), the number of partons which carry a fraction of the nucleon momentum between \( x \) and \( x + dx \). The distribution \( xf(x, \mu^2) \) is then the parton momentum density.\(^3\) 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 (\( \overline{\text{MS}} \) in QCDNUM).\(^4\)

Introducing a short-hand notation for the Mellin convolution,

\[
[f \otimes g](x) = \int_x^1 \frac{dz}{z} f \left( \frac{x}{z} \right) g(z) = \int_x^1 \frac{dz}{z} f(z) \left( \frac{x}{z} \right),
\]

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

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

If the \( x \) dependencies of the parton densities are known at some scale \( \mu_0^2 \), they can be evolved to other values of \( \mu^2 \) 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 [9]:

\[
P_{gq_i} = P_{g\bar{q}_i} = P_{gq} \\
P_{qg} = P_{\bar{q}g} = \frac{1}{2n_f}P_{gg} \\
P_{q_iq_k} = P_{\bar{q}_iq_k} = \delta_{ik}P_{qq}^\nu + P_{qq}^s \\
P_{q_iq_k} = P_{\bar{q}_iq_k} = \delta_{ik}P_{\bar{q}q}^\nu + P_{\bar{q}q}^s.
\]

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

\[
q_s = \sum_{i=1}^{n_f}(q_i + \bar{q}_i)
\]

obeys an evolution equation coupled to the gluon density

\[
\frac{\partial}{\partial \ln \mu^2} \left( \begin{array}{c} q_s \\ g \end{array} \right) = \left( \begin{array}{cc} P_{qq} & P_{qs} \\ P_{gq} & P_{gs} \end{array} \right) \otimes \left( \begin{array}{c} q_s \\ g \end{array} \right),
\]

with \( P_{qq} \) given by

\[
P_{qq} = P_{qq}^\nu + P_{qq}^s + n_f(P_{q\bar{q}}^s + P_{\bar{q}q}^s).
\]

Likewise, we find that the non-singlet combinations

\[
q_{ij}^\pm = (q_i \pm \bar{q}_i) - (q_j \pm \bar{q}_j) \\
q_v = \sum_{i=1}^{n_f}(q_i - \bar{q}_i)
\]

\(^3\)In this section we use the number densities \( f(x, \mu^2) \). In QCDNUM itself, however, we use \( xf(x, \mu^2) \).

\(^4\)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
\[
\frac{\partial q_{ij}^\pm}{\partial \ln \mu^2} = P_\pm \otimes q_{ij}^\pm \quad \text{and} \quad \frac{\partial q_v}{\partial \ln \mu^2} = P_v \otimes q_v, \tag{2.12}
\]
with splitting functions defined by
\[
P_\pm = P_{qq}^v \pm P_{q\bar{q}}^v \quad \text{and} \quad P_v = P_{qq}^v - P_{q\bar{q}}^v + n_f (P_{qs}^v - P_{s\bar{q}}^v). \tag{2.13}
\]
The evolution of the \( q_{ij}^\pm \) 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 \( \alpha_s \) which presently is known up to NNLO. For the four splitting functions \( P_{ij} \) in (2.9) we may write
\[
P_{ij}(x, \mu^2) = a_s(\mu^2) P_{ij}^{(0)}(x) + a_s^2(\mu^2) P_{ij}^{(1)}(x) + a_s^3(\mu^2) P_{ij}^{(2)}(x) + O(a_s^4) \tag{2.14}
\]
where we have set, as in the previous section, \( a_s = \alpha_s / 2\pi \). Note the separation in the variables \( x \) and \( \mu^2 \) on the right-hand side of (2.14). We drop again the arguments \( x \) and \( \mu^2 \) and write the expansion of the non-singlet splitting functions as
\[
P_\pm = a_s P_{qq}^{(0)} + a_s^2 P_{\pm}^{(1)} + a_s^3 P_{\pm}^{(2)} + O(a_s^4) \quad \text{and} \quad P_v = a_s P_{qq}^{(0)} + a_s^2 P_v^{(1)} + a_s^3 P_v^{(2)} + O(a_s^4). \tag{2.15}
\]
Truncating the right-hand side to the appropriate order in \( a_s \), it is seen that at LO the three types of non-singlet obey the same evolution equations. At NLO, \( q_{ij}^- \) and \( q_v \) 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 \( n_f \). This number is set to 3 below \( \mu^2_F = \mu^2_c \) and changed to \( n_f = (4, 5, 6) \) at and above the thresholds \( \mu^2_F = \mu^2_{c, b, t} \). In case \( \mu^2_F \neq \mu^2_R \), QCDNUM adjusts the \( \mu^2_F \) thresholds such that \( n_f \) 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 A. Those at NLO can be found in [12] (non-singlet) and [13] (singlet).\(^5\) The NNLO splitting functions and their parametrisations 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 [14], and references therein. The time-like evolution of fragmentation functions at LO is described in [15], see also Appendix A. The NLO time-like splitting functions can be found in [12] and [13].

### 2.3 Renormalisation Scale Dependence

In the previous section, we have assumed that the factorisation and renormalisation scales are equal. For \( \mu^2_F \neq \mu^2_R \) we expand \( a_s \) in a Taylor series on a logarithmic scale

\(^5\)Two well-known misprints in [13] are: (i) the lower integration limit in the definition of \( S_2(x) \) must read \( x / (1 + x) \); (ii) in the expression for \( \tilde{F}^{(1, T)}_{FF} \) the term \( (10 - 18 x - 16 \frac{x^2}{3}) \) must read \( (-10 - 18x - \frac{16}{3}x^2) \).
around $\mu_R^2$

$$a_s(\mu_F^2) = a_s(\mu_R^2) + a'_s(\mu_R^2)L_R + \frac{1}{2} a''_s(\mu_R^2)L_R^2 + \ldots \quad (2.16)$$

with $L_R = \ln(\mu_F^2/\mu_R^2)$. Using (2.1) to calculate the derivatives in (2.16), we obtain

$$a_s(\mu_F^2) = a_s(\mu_R^2) - \beta_0 L_R a_s^2(\mu_R^2) - (\beta_1 L_R - \beta_0^2 L_R^2) a_s^3(\mu_R^2) + O(a_s^4)$$

$$a_s^2(\mu_F^2) = a_s^2(\mu_R^2) - 2 \beta_0 L_R a_s^3(\mu_R^2) + O(a_s^4)$$

$$a_s^3(\mu_F^2) = a_s^3(\mu_R^2) + O(a_s^4). \quad (2.17)$$

To calculate the renormalisation scale dependence of the evolved parton densities, the powers of $a_s$ 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 $a_s$ when we evolve at LO, to order $a_s^2$ when we evolve at NLO, and to order $a_s^3$ 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

$$|p\rangle = \sum_{i=1}^{n_f} (\alpha_i |q_i\rangle + \beta_i |\bar{q}_i\rangle), \quad (2.18)$$

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

$$\sum_{i=1}^{n_f} (\alpha_i + \beta_i) = 0, \quad (2.19)$$

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

$$|p\rangle = \sum_{i=1}^{n_f} (b_i^+ |q_i^+\rangle + b_i^- |q_i^-\rangle). \quad (2.20)$$

The coefficients $b_i^\pm$, $\alpha_i$ and $\beta_i$ are related by

$$b_i^\pm = \frac{\alpha_i \pm \beta_i}{2}, \quad \alpha_i = b_i^+ + b_i^-, \quad \beta_i = b_i^+ - b_i^- \quad (2.21)$$

We define a basis of singlet, valence, and $2(n_f - 1)$ additional non-singlets by

$$|e_1^+\rangle = |q_0\rangle, \quad |e_1^-\rangle = |\bar{q}_0\rangle, \quad |e_i^\pm\rangle = \sum_{j=1}^{i-1} |q_j^\pm\rangle - (i - 1) |q_i^\pm\rangle \quad \text{for} \quad 2 \leq i \leq n_f. \quad (2.22)$$
In matrix notation, this transformation can be written as

\[ |e^\pm\rangle = U |q^\pm\rangle, \]  

(2.23)

where \( U \) is the \( n_f \times n_f \) sub-matrix of the \( 6 \times 6 \) transformation matrix

\[
U = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 0 & 0 & 0 & 0 \\
1 & 1 & -2 & 0 & 0 & 0 \\
1 & -1 & 1 & -3 & 0 & 0 \\
1 & 1 & 1 & 1 & -4 & 0 \\
1 & 1 & 1 & 1 & 1 & -5 \\
\end{pmatrix}.
\]

(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 \( U \) 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

\[ U^{-1} = U^T S^2, \]  

(2.25)

where \( U^T \) is the transpose of \( U \) and \( S^2 \) is the square of the diagonal scaling matrix:

\[
S^2_{ij} = \delta_{ij} \left( \sum_{k=1}^{n_f} U^2_{ik} \right)^{-1} = \begin{cases} 
\delta_{ij}/n_f & \text{for } i = 1 \\
\delta_{ij}/i(i-1) & \text{for } i > 1.
\end{cases}
\]

(2.26)

Using (2.25) and (2.26) to invert any \( n_f \times n_f \) sub-matrix of (2.24), it is straightforward to show by explicit calculation that

\[
U^{-1}_{ij} = \begin{cases} 
1/n_f & \text{for } j = 1 \\
-1/j & \text{for } j = i \neq 1 \\
1/j(j-1) & \text{for } j > i \\
0 & \text{otherwise}.
\end{cases}
\]

(2.27)

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

\[
|q^\pm_i\rangle = \frac{|e^\pm_i\rangle}{n_f} + \sum_{j=2}^{n_f} \frac{|e^\pm_j\rangle}{j(j-1)} \\
|q^\pm_i\rangle = \frac{|e^\pm_i\rangle}{n_f} - \frac{|e^\pm_i\rangle}{i} + \sum_{j=i+1}^{n_f} \frac{|e^\pm_j\rangle}{j(j-1)}.
\]

(2.28)

We can now write the linear combination \( |p\rangle \) on the \( |e^\pm\rangle \) basis as

\[ |p\rangle = \sum_{i=1}^{n_f} (d^+_i |e^+_i\rangle + d^-_i |e^-_i\rangle), \]

(2.29)

where the coefficients \( d^+_i \) are related to the \( b^+_i \) of (2.20) by

\[ d^+_i = \sum_{j=1}^{n_f} b^+_j U^{-1}_{ji}, \quad b^+_i = \sum_{j=1}^{n_f} d^+_j U_{ji}. \]

(2.30)
Let the starting values of the DGLAP evolutions be given by the gluon density and $2n_f$ arbitrary quark densities, that is, by $2n_f + 1$ functions of $x$ at some input scale $\mu_0^2$. We can arrange the input quark densities in a $2n_f$-dimensional vector $|p\rangle$. Likewise, we store the densities $|q_i^{\pm}\rangle$, the $|e_i^{\pm}\rangle$ in a vector $|e\rangle$ and the $b^{\pm}$ coefficients of each input density in the rows of a $2n_f \times 2n_f$ matrix $B$. The flavour decomposition of the input densities can then be written as $|p\rangle = B|q\rangle$ and the singlet/non-singlet decomposition as $|e\rangle = TB^{-1}|p\rangle$. Provided that $B^{-1}$ 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 $|e\rangle = TB^{-1}|p\rangle$.

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

\begin{equation}
2.32
|e\rangle = TB^{-1}|p\rangle.
\end{equation}

### 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 $\mu^2$, with $3 \leq n_f \leq 6$. In the FFNS, the input scale $\mu_0^2$ can be chosen anywhere within the boundaries of the evolution grid, although one should be careful with backward evolution in QCDNUM; see Section 3.3.

In the VFNS, the number of flavours changes from $n_f$ to $n_f + 1$ when the factorisation scale is equal to the pole mass of the heavy quarks $\mu^2_h = m^2_h$, $h = (c, b, t)$. A heavy quark $h$ is thus considered to be infinitely massive below $\mu^2_h$ and mass-less above $\mu^2_h$. 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 $\mu^2_h$. Such an abrupt turn-on at a fixed scale is of course unphysical but this poses no problem since the parton densities themselves are not observables. 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 [16, 17, 18]. Such calculations are not part of QCDNUM itself, but can be coded in add-on packages; see Section 6.

An important feature of VFNS evolution is that the input scale $\mu_0^2$ cannot be above the lowest heavy flavour threshold $\mu^2_c$. 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 $\alpha_s$ 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 $\alpha_s$ evolution is discontinuous when crossing a threshold in the VFNS. Beyond LO there are not only discontinuities in the slope but also in $\alpha_s$ itself [11]. In N^4LO, the value of
The discontinuities in the basis vectors found in Appendix B of \cite{20}. Here

\[ a_s^{(n_f+1)}(\kappa \mu_h^2) = a_s^{(n_f)}(\kappa \mu_h^2) + \sum_{n=1}^{\ell} \left\{ a_s^{(n_f)}(\kappa \mu_h^2) \right\}^{n+1} \sum_{j=0}^{n} C_{n,j} \ln^{j} \kappa \} \quad \ell = 1, 2. \quad (2.33) \]

Here \( \mu_h^2 \) is the threshold defined on the factorisation scale and \( \kappa \) is the ratio \( \mu_h^2/\mu_F^2 \) at \( \mu_h^2 \).

For \( a_s = a_s/4\pi \), the coefficients \( C \) in (2.33) read

\[ C_{1,0} = 0, \quad C_{1,1} = \frac{2}{3}, \quad C_{2,0} = \frac{14}{3}, \quad C_{2,1} = \frac{38}{3}, \quad C_{2,2} = \frac{4}{9}. \]

Note that there is always a discontinuity in \( a_s \) 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, \( a_s^{(n_f+1)} \) is computed directly from (2.33) while for downward evolution, \( a_s^{(n_f-1)} \) is evaluated by numerically solving the equation

\[ a_s^{(n_f)} - a_s^{(n_f-1)} - \Delta a_s(a_s^{(n_f-1)}) = 0, \]

where the function \( \Delta a_s(a_s) \) is given by the second term on the right-hand side of (2.33).

In the VFNS at NNLO, not only \( a_s \) but also the parton densities have discontinuities at the flavour thresholds \cite{20}:

\[
g(x, \mu_h^2, n_f + 1) = g(x, \mu_h^2, n_f) + \Delta g(x, \mu_h^2, n_f)\]

\[
q_i^\pm(x, \mu_h^2, n_f + 1) = q_i^\pm(x, \mu_h^2, n_f) + \Delta q_i^\pm(x, \mu_h^2, n_f) \quad i = 1, \ldots, n_f
\]

\[
h^+(x, \mu_h^2, n_f + 1) = \Delta h^+(x, \mu_h^2, n_f)
\]

\[
h^-(x, \mu_h^2, n_f + 1) = \Delta h^-(x, \mu_h^2, n_f) = 0, \quad (2.34)
\]

where \( h = (c, b, t) \) for \( n_f = (3, 4, 5) \). Note that a heavy quark \( h \) becomes a light quark \( q_i \) above the threshold \( \mu_h^2 \).

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. \cite{20} vanish). So we may write

\[
\Delta g(x, \mu_h^2, n_f) = a_s^2 \left\{ [A_{gg} \otimes q_s](x, \mu_h^2, n_f) + [A_{gg} \otimes g](x, \mu_h^2, n_f) \right\}
\]

\[
\Delta q_i^\pm(x, \mu_h^2, n_f) = a_s^2 \left\{ [A_{qq} \otimes q_i^\pm](x, \mu_h^2, n_f) \right\}
\]

\[
\Delta h^+(x, \mu_h^2, n_f) = a_s^2 \left\{ [A_{hg} \otimes q_s](x, \mu_h^2, n_f) + [A_{hg} \otimes g](x, \mu_h^2, n_f) \right\}. \quad (2.35)
\]

Here \( a_s \) stands for \( a_s^{(n_f+1)}(\kappa \mu_h^2) \) as defined by (2.33). The convolution kernels \( A_{ij} \) can be found in Appendix B of \cite{20}.\(^6\)

The discontinuities in the basis vectors \( |e_i^\pm \rangle \) are calculated from

\[
e_i^\pm(x, \mu_h^2, n_f + 1) = e_i^\pm(x, \mu_h^2, n_f) + \Delta e_i^\pm(x, \mu_h^2, n_f) + \lambda_i(n_f) \Delta h^+(x, \mu_h^2, n_f), \quad (2.36)
\]

\( ^6\)In the notation of \cite{20}, \( A_{gg} = A_{gg}^{S,(2)} \) (eq. B.5), \( A_{gg} = A_{gg}^{S,(2)} \) (B.7), \( A_{qg} = A_{qg}^{NS,(2)} \) (B.4), \( A_{hq} = A_{hq}^{PS,(2)} \) (B.1) and \( A_{hg} = A_{hg}^{S,(2)} \) (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 \( \lambda_i(n_f) \) are

\[
\begin{array}{c|cccccc}
 n_f & \lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 & \lambda_5 & \lambda_6 \\
3 & 1 & 0 & 0 & -3 \\
4 & 1 & 0 & 0 & 0 & -4 \\
5 & 1 & 0 & 0 & 0 & 0 & -5 \\
\end{array}
\] (2.37)

When the densities are evolved upward in \( \mu^2 \), it is straightforward to calculate with (2.34) and (2.35) the parton densities at \( n_f + 1 \) from those at \( n_f \). However, QCDNUM is capable to invert the relation between \( n_f \) and \( n_f + 1 \) 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)

\[
\left( \begin{array}{c} q_s \\ g \end{array} \right)^{(n_f+1)} = \left( \begin{array}{c} q_s \\ g \end{array} \right)^{(n_f)} + a_s^2 \left( \begin{array}{cc} A_{qq} + A_{hq} \\ A_{sq} \\ A_{gg} \end{array} \right) \otimes \left( \begin{array}{c} q_s \\ g \end{array} \right)^{(n_f)}. \] (2.38)

In Section 3.3 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 \( \mu^2 \), because the heavy flavours are simply set to zero below their thresholds, instead of being evolved. The evolution of the set \( |e_i^+\rangle \) thus proceeds in the VFNS as follows: The singlet/valence densities \( |e_1^+\rangle \) and the light non-singlets \( |e_{2,3}^+\rangle \) are evolved both upward and downward starting from some scale \( \mu_0^2 < \mu_c^2 \). The heavy non-singlets \( |e_{4,5,6}^+\rangle \) are dynamically generated from the DGLAP equations by upward evolution from the thresholds \( \mu_{c,b,t}^2 \). 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 \( q_s(x, \mu_s^2) \), for charm (full curve) and bottom (dotted curve).](image)
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 $x \approx 10^{-2}$.

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 $\alpha_s$ 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 $\beta$-functions, as is required in some heavy flavour calculations, see for instance [21].

3 Numerical Method

The DGLAP evolution equations are in QCDNUM numerically solved on a discrete $n \times m$ grid in $x$ and $\mu^2$. 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’ [19], 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 $\mu^2$. The order of the $x$-interpolation can in be set to $k = 2$ (linear) or 3 (quadratic). The interpolation in $\mu^2$ 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 $\mu_0^2$ to any other scale $\mu_i^2$ 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, [22, 23].

3.1 Polynomial Spline Interpolation

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

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

and parameterise it in each interval by a piecewise polynomial of order $k$. Such a piecewise polynomial is turned into a spline by imposing one or more continuity relations

7In 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.
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 piecewise polynomials.

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

\[
h(y) = \sum_i A_i Y_i(y).
\]  

(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 [24].

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

\[ 0 1 2 3 4 \]
\[ 0 \]
\[ 0.5 \]
\[ 1 \]
\[ \text{Graph (c)} \]
\[ Y1 \]
\[ Y2 \]
\[ Y3 \]
\[ Y4 \]
\[ 0 1 2 3 4 \]
\[ 0 \]
\[ 0.5 \]
\[ 1 \]
\[ \text{Graph (b)} \]
\[ Y1 \]
\[ Y2 \]
\[ Y3 \]
\[ 0 1 2 3 4 \]
\[ 0 \]
\[ 0.5 \]
\[ 1 \]
\[ \text{Graph (a)} \]
\[ Y1 \]
\[ Y2 \]
\[ Y3 \]
\[ Y4 \]

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 \( h'(y_0) = 0 \).\(^8\) With these boundary conditions—and because

\(^8\)A parton density parameterisation should thus behave like \( h(y \to 0) \propto y^\lambda \) with \( \lambda > 1 \) 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, ..., \( n \) 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):

\[
Y_i(y) = Y_i(y - y_{i-1}). \tag{3.2}
\]

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

\[
h(y_0) = 0 \]
\[
h(y_i) = A_i Y_i(y_i) = A_i \quad 1 \leq i \leq n. \tag{3.3}
\]

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

\[
h(y_0) = 0 \]
\[
h(y_1) = A_1 Y_1(y_1) = A_1/2 \]
\[
h(y_i) = A_{i-1} Y_{i-1}(y_i) + A_i Y_i(y_i) = (A_{i-1} + A_i)/2 \quad 2 \leq i \leq n. \tag{3.4}
\]

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

\[
h = S a \tag{3.5}
\]

where \( S \) 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 \( h = (h_1, \ldots, h_4)^T \) and the matrix

\[
S = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{with inverse} \quad S^{-1} = 2 \begin{pmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{pmatrix}. \tag{3.6}
\]

Note that \( S \) is sparse but \( S^{-1} \) is not. Thus, when a parton distribution \( h_0 \) is given at some input scale \( \mu_0^2 \), the corresponding vector \( a_0 \) of spline coefficients is found by solving (3.5).\(^9\) This vector is then evolved to other values of \( \mu^2 \) using the DGLAP evolution equations as is described in the next two sections.

\(^9\)Obtaining \( a \) from solving (3.5) by forward substitution (Appendix B) costs \( O(2n) \) operations. This is cheaper than the alternative of calculating \( a = S^{-1} h \) which costs \( O(n^2/2) \) 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 \( p = x f \) and a kernel \( q = x g \). 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 \( p = x f \), \( q = x g \) and \( r = x h \) 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:

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

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

\[
\frac{\partial h(y, t)}{\partial t} = \int_0^y dz \ Q(z, t) h(y - z, t) = \int_0^y dz \ Q(y - z, t) h(z, t) \tag{3.10}
\]

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

\[
I(y, t) \equiv \int_0^y dz \ Q(y - z, t) h(z, t). \tag{3.11}
\]

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)

\[
I(y_i) = \sum_{j=1}^i A_j \int_0^{y_i} dz \ Q(y_i - z) Y_j(z) = \sum_{j=1}^i W_{ij} A_j \quad (1 \leq i \leq n). \tag{3.12}
\]

The summation is over the first \( i \) terms only, because B-splines with an index \( j > i \) are zero in the integration domain \( z \leq y_i \), 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

\[
W_{ij} = \int_{y_{j-1}}^{y_i} dz \ Q(y_i - z) Y_j(z) = \int_0^{y_i - y_{j-1}} dz \ Q(y_i - y_{j-1} - z) Y_1(z) \tag{3.13}
\]

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):

$$W_{ij} = w_{i - j + 1} \quad \text{with} \quad w_{\ell} \equiv \int_0^{y_{\ell}} dz \, Q(y_{\ell} - z) Y_1(z) \quad (1 \leq \ell \leq n). \quad (3.14)$$

The integrand only contributes in the region $k\Delta$ where $Y_1$ is non-zero so that in practical calculations the upper integration limit $y_{\ell}$ is replaced by $\min(y_{\ell}, k\Delta)$, with $\Delta$ the grid spacing. We remark that the calculation of the weights $w_{\ell}$ 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 A.

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

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

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

$$W(t) = a_s(t)\{ W^{(0)} + a_s(t)W^{(1)} + \cdots \} \quad (3.16)$$

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

$$I = Wa. \quad (3.17)$$

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

$$W = W_a S^{-1} W_b. \quad (3.18)$$

Here $W_a$ and $W_b$ are the weight tables of $K_a$ and $K_b$, respectively, and $S$ 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’ [25] (times $x$) is calculated from the Fourier convolution

$$I(y_i) = \int_0^{y_i} dz \, h_a(z) h_b(y_i - z). \quad (3.19)$$
Inserting the spline representation (3.1) gives an expression for the convolution integral as a weighted sum over the set of spline coefficients \( a \) of \( h_a \) and \( b \) of \( h_b \),

\[
I(y_i) = \sum_{j=1}^{i} \sum_{k=1}^{i} A_j B_k \ W_{ijk} \quad \text{with} \quad W_{ijk} = \int_0^{y_i} dz \ Y_j(z)Y_k(y_i-z).
\]

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

\[
W_{ijk} = \int_0^{y_i-j+1} dz \ Y_1(z)Y_k(y_i-j+1-z).
\]

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

\[
W_{ijk} = \int_0^{y_i-j-k+2} dz \ Y_1(z)Y_1(y_i-j-k+2-z).
\]

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

\[
I(y_i) = \sum_{j=1}^{i} \sum_{k=1}^{i-j+1} A_j B_k \ w_{i-j-k+2} \quad \text{with} \quad w_\ell = \int_0^{y_\ell} dz \ Y_1(z) Y_1(y_\ell-z). \quad (3.20)
\]

Because \( Y_1 \) has a limited support, it turns out that only the first 3 (5) terms of \( w_\ell \) 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 DGLAP Evolution

We denote by the vector \( h_0 \) a non-singlet quark density at the input scale \( t_0 = \ln \mu_0^2 \). The derivative of \( h_0 \) 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)

\[
\frac{dh_0}{dt} = \frac{dS a_0}{dt} = W_0 a_0 \quad \text{or} \quad \frac{da_0}{dt} = S^{-1} W_0 a_0. \quad (3.21)
\]

Likewise we have at \( t_1 \)

\[
a'_1 = S^{-1} W_1 a_1. \quad (3.22)
\]

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

Assuming that \( a(t) \) is quadratic in \( t \), we can relate \( a_0, a_1, a'_0 \) and \( a'_1 \) by

\[
a_1 = a_0 + (a'_0 + a'_1) \Delta_1 \quad \text{with} \quad \Delta_1 = (t_1 - t_0)/2.
\]

If \( t_1 > t_0 \), \( \Delta_1 \) is positive and we perform forward evolution. If \( t_1 < t_0 \), \( \Delta_1 \) is negative and we perform backward evolution.

Inserting (3.21) and (3.22) in (3.23) we obtain a relation between the known spline coefficients \( a_0 \) and the unknown coefficients \( a_1 \)

\[
(1 - S^{-1}W_1 \Delta_1) \ a_1 = (1 + S^{-1}W_0 \Delta_1) \ a_0. \quad (3.24)
\]
Multiplying both sides from the left by $U_1 \equiv S/\Delta_1$ gives

$$ (U_1 - W_1) a_1 = (U_1 + W_0) a_0. \quad (3.25) $$

Eq. (3.25) is more convenient than (3.24) because matrix multiplication $S^{-1}W$ is replaced by matrix addition.\(^\text{10}\) Note that $U$ is a lower diagonal band matrix so that $U \pm W$ 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 $a_0$:

1. At $t_0$, calculate $a_0$ from (3.5), $W_0$ from (3.16) and $U_1$ as defined above. Then construct the vector $b_1 \equiv (U_1 + W_0) a_0$.

2. Subsequently, at $t_1$,
   
   (a) Calculate $W_1$ and the lower triangular matrix $V_1 = U_1 - W_1$;
   
   (b) Solve the equation $V_1 a_1 = b_1$ by forward substitution, see Appendix B;
   
   (c) Calculate $U_2$ and $b_2 = (U_1 + U_2) a_1 - b_1$ for the next evolution to $t_2$.\(^\text{11}\)

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 $O(n)$ and solving one triangular matrix equation which has an operation count $O(n^2/2)$. 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 ($a_s$) and gluon ($a_g$) spline coefficients, provided we make the following replacements in the formalism:

$$ a \rightarrow \begin{pmatrix} a_s \\ a_g \end{pmatrix}, \quad S \rightarrow \begin{pmatrix} S \\ S \end{pmatrix}, \quad W \rightarrow \begin{pmatrix} W_{qq} & W_{qs} \\ W_{sg} & W_{gg} \end{pmatrix}. \quad (3.26) $$

In Appendix B is shown how the coupled triangular equations are solved by extending the forward substitution algorithm. The operation count is $4 \times O(n^2/2)$ so that for $m$ grid points in $t$ we have in total $O(2n^2m)$ operations for the singlet-gluon evolution and $O(n^2m/2)$ 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 $n_f + 1$ as given by (2.38) can be written as

$$ \begin{pmatrix} S \\ S \end{pmatrix} \begin{pmatrix} a_s \\ a_g \end{pmatrix}^{(n_f+1)} = \begin{pmatrix} S + A_{qq} + A_{hq} & A_{hg} \\ A_{gq} & S + A_{gg} \end{pmatrix} \begin{pmatrix} a_s \\ a_g \end{pmatrix}^{(n_f)}. \quad (3.26) $$

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

$$ S a_{ns}^{(n_f+1)} = (S + A_{qq}) a_{ns}^{(n_f)} + \lambda \left( A_{hq} a_{s}^{(n_f)} + A_{hg} a_{g}^{(n_f)} \right), \quad (3.27) $$

\(^{10}\)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.

\(^{11}\)Using (3.25) it is a simple exercise to establish this relation between $b$, $U$ and $a$. Note that $b$ in step (2c) is calculated much faster than $b$ in step (1).
where \( \lambda \) is defined by (2.36). Also this equation can easily be inverted.

It can be seen from (3.5) and (3.23) 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 \( n_f - 1 \) 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 \( h(y,t_{c,b,t} - \epsilon) \) to view the discontinuity.\(^{12}\)

In \textsc{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 \( \Delta \), 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.

As remarked above, the evolution algorithm can—at least in principle—handle both forward and backward evolution in \( \mu^2 \) simply by changing the sign of \( \Delta \) in (3.23). 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 \textsc{qcdnum} this numerical instability is handled as follows: (i) evolve downward from \( \mu_0^2 \) to \( \mu^2 \) in the linear interpolation scheme (which is stable); (ii) then take \( \mu^2 \) as the starting scale and evolve \textit{upward} to \( \mu_0^2 \) in the quadratic interpolation scheme (also stable); (iii) calculate the difference \( \Delta f \) between the newly evolved pdf and the original one at \( \mu_0^2 \); (iv) subtract \( \Delta f \) from the starting value at \( \mu_0^2 \) 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 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 (\textsc{qcdnum} default), perhaps two, are sufficient while more iterations

\(^{12}\)Do not take \( \epsilon \) too small because \textsc{qcdnum} may snap to the threshold, see Section 5.2.
tend to spoil the convergence. Clearly best is to limit the range of downward evolution by keeping \( \mu_0^2 \) 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 \( u = Da \), where \( D \) 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 \( v = Ea \), where \( E \) is the lower diagonal Toeplitz band matrix \((\frac{1}{2}b_2, b_2, \frac{1}{2}b_2)\).

The maximum deviation \( \epsilon = \|u - v\| = \|(D - E)a\| \) should be small; for pdfs sampled on a reasonably dense grid, \( \epsilon \approx 0.1 \) or less. For each pdf evolution, \( \epsilon \) is computed at the input scale, and at the lower and upper end of the \( \mu^2 \) 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.

Figure 4: A non-singlet parton density \( xf(x) \) versus \( \log(x) \) evolved downward from \( \mu_0^2 = 5 \) to \( \mu^2 = 2 \text{ GeV}^2 \) 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.
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 E). Sub-directories contain the source code, example jobs, write-up and a simple script to make a QCDNUM library, see the README file. 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 C.3 and D.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:

\begin{verbatim}
mxg0  Maximum number of multiple x-grids [5].
mxx0  Maximum number of points in the x-grid [300].\footnote{For technical reasons the maximum number of grid points is about 10 less than mxx0 and mqq0.}
mqq0  Maximum number of points in the \(\mu^2\)-grid [150].\footnote{For technical reasons the maximum number of grid points is about 10 less than mxx0 and mqq0.}
mpt0  Maximum number of interpolation points in the fast convolution engine [5000].
miw0  Maximum number of information words in a weight store [20].
mbf0  Maximum number of fast convolution scratch buffers [20].
nwf0  Size of the QCDNUM dynamic store in words [1200000].
\end{verbatim}

The first 6 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 \(x-\mu^2\) grid. It also depends on how many different sets of tables (un-polarised pdfs, polarised pdfs, fragmentation functions, etc.) you want to store. In this respect, QCDNUM is very user-friendly by always gracefully grinding to a halt if it runs out of memory, with a message that tells how large nwf0 should be.

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.
**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 \(\mu^2\) and \(\alpha_s(m^2_Z)\) is calculated.
The first step in a QCDNUM based analysis is initialisation (\texttt{qcinit}), setting up the $x$-$\mu^2$ grid (\texttt{gmake, gqmake}) and the calculation of the weight tables (\texttt{fillwt}). The weights depend on the grid definition and the interpolation order so that \texttt{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 $3 \leq n_f \leq 6$ so that you do not have to call \texttt{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 (\texttt{setord}) and the input value of $\alpha_s$ at some renormalisation scale $\mu^2_0$ (\texttt{setalf}). The call to \texttt{setcbt} sets the \texttt{vfns} mode and defines the thresholds on the factorisation scale $\mu^2_F$. All the calls that set evolution parameters are destructive in the sense that they invalidate the parton densities in memory, if any. In this way all QCDNUM results are consistently obtained with the same value of $\alpha_s$, the same perturbative order, etc.

The second step is to evolve the parton densities from input specified at the scale $\mu^2_0$. It is important to note that QCDNUM evolves parton \textit{momentum} densities $x f(x)$, although all theory in this write-up is expressed in terms of parton \textit{number} densities $f(x)$. The evolution is done by calling the routine \texttt{evolfg} which evolves $2n_f + 1$ 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 $\mu^2_0$ must lie below the charm threshold $\mu^2_c$ so that, as a consequence, $\mu^2_c$ must lie above the lower boundary of the $\mu^2$ grid.

The flavour composition of each of the input quark densities is given by a table of weights \texttt{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 $d, u, s, \ldots$ and not $u, d, s, \ldots$ as often is the case in other programs.

The $x$ dependence of these momentum densities at $\mu^2_0$ must be coded for each identifier in an if-then-else block in the function \texttt{func}. The sum rules

$$
\int_0^1 xg(x)dx + \int_0^1 xq^s(x)dx = 1, \\
\int_0^1 [d(x) - \bar{d}(x)]dx = 1, \\
\int_0^1 [u(x) - \bar{u}(x)]dx = 2
$$

(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 $\mu^2_0$. The easiest way to evolve with a symmetric strange sea is to include $x s_v = x(s - \bar{s})$ in the collection of input densities and set it to zero for all $x$ at the input scale $\mu^2_0$. In the VFNS at LO or NLO, the generated heavy
flavour densities \( h = (c, b, t) \) 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 \( \mu^2 \). Also here the flavours \( d, u, s, \ldots \) are indexed according to the PDG convention. The last call in the example program evolves the input value of \( \alpha_s \) to the scale \( m_Z^2 \). This evolution is completely stand-alone and does not make use of the \( \mu^2 \) 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.

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 \( \mu^2 \) grid may then map onto values of \( \mu_R^2 < \Lambda^2 \).

Another QCDNUM feature is that \( n_f = (4, 5, 6) \) and not \((3, 4, 5)\) at the heavy flavour thresholds \( \mu_h^2 \). This implies, first of all, that parton evolution in the VFNS must start from \( \mu_0^2 < \mu_c^2 \) and not from \( \mu_0^2 \leq \mu_c^2 \), simply because the number of flavours must be \( n_f = 3 \) at the starting scale. There is, however, no restriction on the starting (renormalisation) scale of \( \alpha_s \) 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 \( \alpha_s \) 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 quadratically with the number of grid points in \( x \). With linear (quadratic) interpolation the accuracy increases linearly (quadratically) 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 \( \bar{N} \)-space evolution program PEGASUS [19]. In this comparison a default set of initial distributions [26] is evolved at NNLO from \( \mu^2 = 2 \) to \( \mu^2 = 10^4 \, \text{GeV}^2 \) with \( n_f = 4 \) flavours. The dashed curve in the top plot of Figure 6 shows 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 3.3) as shown by the full curve in the top plot of Figure 6. 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 6 that the precision is
Figure 6: 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 dotted curves in Figure 6 (top and bottom) correspond to evolution with quadratic 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 6. Note that this is for evolution up to $\mu^2 = 10^4$ GeV$^2$; at lower $\mu^2$ the accuracy is even better since it increases (roughly linearly) with decreasing $\ln(\mu^2)$. To fully validate the QCDNUM evolution with

<table>
<thead>
<tr>
<th>$n$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear interpolation</td>
<td>200</td>
<td>$10^{-5}$</td>
<td>0.01</td>
<td>0.10</td>
<td>0.40</td>
</tr>
<tr>
<td>Quadratic interpolation</td>
<td>100</td>
<td>$10^{-5}$</td>
<td>0.20</td>
<td>0.40</td>
<td>0.60</td>
</tr>
<tr>
<td>Relative point density</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>
we have made additional comparisons in the FFNS with $n_f = 3, 5$ 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 3.3, 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 $\mu^2$ 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 available QCDNUM subroutines and functions. For convenience a list of these is given in Table 2. 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:

\[
\begin{align*}
ix &= \text{ixfrmx}(x) \quad \text{! ok} \\
ix &= \text{ixfrmx}(0.1D0) \quad \text{! ok} \\
ix &= \text{ixfrmx}(0.1) \quad \text{! wrong!}
\end{align*}
\]

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:

\[
\begin{align*}
\text{write}(6,*), \text{fvalxq}(1,0,x,q,1) \quad \text{! not recommended}
\end{align*}
\]

\[
\begin{align*}
glue &= \text{fvalxq}(1,0,x,q,1) \quad \text{! OK} \\
\text{write}(6,*), \text{glue} \quad \text{! OK}
\end{align*}
\]

\footnote{Similar benchmarking between HOPPET [22] and PEGASUS is given in [26] and [27], 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>
<thead>
<tr>
<th>Subroutine or function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QCINIT ( lun, 'filename' )</td>
<td>initialise</td>
</tr>
<tr>
<td>SETLUN ( lun, 'filename' )</td>
<td>redirect output</td>
</tr>
<tr>
<td>SET</td>
<td>GETVAL ( 'opt', val )</td>
</tr>
<tr>
<td>SET</td>
<td>GETINT ( 'opt', ival )</td>
</tr>
<tr>
<td>GXMAKE ( xmi, iwt, n, nxin, *nxout, iord )</td>
<td>define x grid</td>
</tr>
<tr>
<td>IXFRMIX ( x )</td>
<td>get i_x from x</td>
</tr>
<tr>
<td>XXATIX ( x, ix )</td>
<td>verify grid point</td>
</tr>
<tr>
<td>QMMAKE ( qarr, wt, n, nqin, *nqout )</td>
<td>define ( \mu^2 ) grid</td>
</tr>
<tr>
<td>IQFRMQ ( q2 )</td>
<td>get ( \mu ) from ( \mu^2 )</td>
</tr>
<tr>
<td>QFRMQ ( iq )</td>
<td>get ( \mu^2 ) from ( \mu )</td>
</tr>
<tr>
<td>QQATIQ ( q2, iq )</td>
<td>verify grid point</td>
</tr>
<tr>
<td>GRPARS ( *nx, *x1, *x2, *nq, *q1, *q2, *io )</td>
<td>get grid definitions</td>
</tr>
<tr>
<td>GXCOPY ( *array, n, *nx )</td>
<td>copy x grid</td>
</tr>
<tr>
<td>GQCOPY ( *array, n, *nq )</td>
<td>copy ( \mu^2 ) grid</td>
</tr>
<tr>
<td>FILLWT ( itype, *idmi, *idma, *nw )</td>
<td>fill weight tables</td>
</tr>
<tr>
<td>FALLWC ( mysub, *idmi, *idma, *nw )</td>
<td>custom weights</td>
</tr>
<tr>
<td>DMPWGT ( itype, lun, 'filename' )</td>
<td>dump weight tables</td>
</tr>
<tr>
<td>READWT ( lun, 'fn', *idmi, *idma, *nw, *ie )</td>
<td>read weight tables</td>
</tr>
<tr>
<td>NWUSED ( *nwtot, *nwuse, *nwtab )</td>
<td>memory words used</td>
</tr>
<tr>
<td>SET</td>
<td>GETORD ( iord )</td>
</tr>
<tr>
<td>SET</td>
<td>GETALF ( alfs, r2 )</td>
</tr>
<tr>
<td>SETCBT ( nfix, iqc, iqb, iqt )</td>
<td>set ( n_f ) or thresholds</td>
</tr>
<tr>
<td>MIXFNS ( nfix, r2c, r2b, r2t )</td>
<td>set MFNS parameters</td>
</tr>
<tr>
<td>GETCBT ( *nfix, *q2c, *q2b, *q2t )</td>
<td>get ( n_f ) or thresholds</td>
</tr>
<tr>
<td>SET</td>
<td>GETABR ( ar, br )</td>
</tr>
<tr>
<td>FFROMF ( fscale )</td>
<td>convert ( \mu^2 ) to ( \mu^2_R )</td>
</tr>
<tr>
<td>FFROMR ( rscale )</td>
<td>convert ( \mu^2_R ) to ( \mu^2 )</td>
</tr>
<tr>
<td>SET</td>
<td>GETCUT ( xmi, q2mi, q2ma, dummy )</td>
</tr>
<tr>
<td>LPASSC ( x, qmu2, *ifail, ichk )</td>
<td>true if inside cuts</td>
</tr>
<tr>
<td>ASFUNC ( r2, *nf, *ierr )</td>
<td>evolve ( \alpha_s(\mu^2_R) )</td>
</tr>
<tr>
<td>EVOLFG ( itype, func, def, iq0, *eps )</td>
<td>evolve all pdfs</td>
</tr>
<tr>
<td>PDFINP ( subr, iset, offset, *epsi, *nwds )</td>
<td>pdfs from outside</td>
</tr>
<tr>
<td>CHKPDF ( iset )</td>
<td>true if pdf set exists</td>
</tr>
<tr>
<td>FVALX</td>
<td>IJ ( iset, id, x</td>
</tr>
<tr>
<td>FPDFX</td>
<td>IJ ( iset, x</td>
</tr>
<tr>
<td>FSUMX</td>
<td>IJ ( iset, def, x</td>
</tr>
<tr>
<td>FSNSX</td>
<td>IJ ( iset, id, x</td>
</tr>
<tr>
<td>SPLCHK ( iset, id, iq )</td>
<td>check spline</td>
</tr>
<tr>
<td>FSPLNE ( iset, id, x, iq )</td>
<td>spline interpolation</td>
</tr>
<tr>
<td>PDFLST ( iset, def, x, qmu2, *pdf, n, ichk )</td>
<td>fast pdf interpolation</td>
</tr>
<tr>
<td>PDFTAB ( iset, def, x, nx, q, nq, *f, ichk )</td>
<td>fast pdf interpolation</td>
</tr>
</tbody>
</table>

Output arguments are pre-fixed with an asterisk (*).
5.1 Initialisation

```fortran
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.

```fortran
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`.

```fortran
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 $|x - y| < \epsilon$, which QCDNUM uses to decide if $x$ and $y$ are equal. Default, `epsi = 1.D-9`.
- `'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 $\alpha_s$ exceeds the limit, a fatal error condition is raised. Default, `alim = 10`\textsuperscript{15}.
- `'qmin'` Smallest possible lower boundary of the $\mu^2$ grid. Default, `qmin = 0.1\text{ GeV}^2`.
- `'qmax'` Largest possible upper boundary of the $\mu^2$ grid. Default, `qmax = 1.D11\text{ GeV}^2`.

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

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

Set or get QCDNUM integer parameters.

- `'iter'` Set the number of iterations in the backward evolution. When set negative, you will evolve backward in the same interpolation scheme as the forward evolution (not recommended). When set to zero, you will evolve backward in the linear interpolation scheme, without iterations (not recommended either). A value larger than zero gives the number of iterations to perform. Default, `iter = 1`. This parameter is only relevant when you work with quadratic splines.

\textsuperscript{15}When you raise `alim > 10` then $\alpha_s$ will at some point be limited by internal cuts in QCDNUM.
‘ntab’ Number of scratch buffers (maximum 20) generated by fastini (fast convolution engine, Section 6.5). Will have no effect when set after the call to fastini. Default, ntab = 5. If you want to generate more than 20 buffers, the value of mbf0 in qcdnum.inc should be increased, and QCDNUM re-compiled.

And then for getint only:

‘vers’ Returns the current 6-digit version number like 170005. Can be used to check if QCDNUM is initialised because the version is set to 0 if not.

‘lunq’ Retrieve the qcdnum logical unit number. Useful if you want to write messages on the same output stream as QCDNUM.

‘mxg0’ . . . ‘nwf0’ Value of a fixed parameter in qcdnum.inc (see Section 4.1).

5.2 Grid Definition

A proper choice of the grid in $x$ and $\mu^2$ is important because it determines the speed and accuracy of the QCDNUM calculations.\(^ {16} \) The grid definition also sets the size of the internal store which contains the weight tables and the tables of parton densities.

The $x$ grid must be strictly equidistant in the variable $y = -\ln x$ but in QCDNUM you can generate multiple equidistant grids (Section 3.3) 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 $\mu^2$ 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 $\mu^2$.

\begin{verbatim}
call GXMAKE ( xmin, iwt, n, nxin, *nxout, iord )
\end{verbatim}

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.

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.

\(^ {16} \) We refer to Section 4.3 for recommended grids in the linear and quadratic interpolation schemes.
**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

```fortran
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

```fortran
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.

```fortran
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

```fortran
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, $\epsilon = 10^{-9}$.

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

Generate a logarithmic $\mu^2_F$ grid on which the parton densities are evolved.\(^\text{17}\)

**qarr**  Input array containing $n$ values of $\mu^2$ 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 $\mu^2_0$, or thresholds $\mu^2_{c,b,t}$.

\(^\text{17}\)Note that $\alpha_s$ is evolved (without using a grid) on $\mu^2_R$ which may or may not be different from $\mu^2_F$. \(33\)
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 $0.1 \leq w \leq 10$. With these weights, you can generate a grid with higher density at low values of $\mu^2$ where $\alpha_s$ is changing rapidly.

n The number of values specified in qarr and wgt ($n \geq 2$).

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 $\mu^2$ 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 $\mu^2$. The inverse function is qfrmiq. To verify that $\mu^2$ 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.

```
```

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 $\mu^2$ grid.

qmi Lower boundary of the $\mu^2$ grid.

qma Upper boundary of the $\mu^2$ grid.

iord Order of the spline interpolation ($2 = \text{linear}, 3 = \text{quadratic}$).

```
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 $\mu^2$ grid.
5.3 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 \texttt{qcdnum} settings. Tables can be created for un-polarised pdfs, polarised pdfs and fragmentation functions. All these pdf types can exist simultaneously in memory. For each type, one gluon table and 12 quark tables are generated by the routines, in addition to the weight tables.

\begin{verbatim}
call FILLWT ( itype, *idmin, *idmax, *nwds )
\end{verbatim}

Partition the pdf store and fill the weight tables used in the calculation of the convolution integrals. Both the \( x \) and \( \mu^2 \) grid must have been defined before the call to \texttt{fillwt}.

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

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

\begin{verbatim}
call fillwt(1,idmin,idmax,nw) !Unpolarised pdfs
call fillwt(2,idmin,idmax,nw) !Polarised pdfs
\end{verbatim}

makes both the un-polarised and the polarised pdfs available. For each pdf type, \texttt{fillwt} creates 13 pdf tables. If there is not enough space in memory to hold all the tables, \texttt{fillwt} returns with an error message telling how much memory it needs. You should then increase the value of \texttt{nwf0} in the include file \texttt{qcdnum.inc}, and recompile \texttt{QCDNUM}.

Note that \texttt{fillwt} acts as a do-nothing when the pdf type already exists in memory:

\begin{verbatim}
call fillwt(1,idmin,idmax,nw) !Unpolarised pdfs
call fillwt(1,idmin,idmax,nw) !Do nothing
\end{verbatim}

\begin{verbatim}
call DMPWGT( itype, lun, 'filename' )
\end{verbatim}

Dump the weight tables (not the pdf tables) of a given pdf type to disk. When \texttt{itype} = 0, all pdf types in memory are dumped.\footnote{This you may have to repeat several times because \texttt{fillwt} proceeds in stages and will report the memory needs of the current stage, but not beyond.} Fatal error if \texttt{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 unformatted so that the output file cannot be exchanged across machines.

\footnote{This does not include the weight tables of custom evolution (\texttt{itype} = 4, see Section 6.6). Such tables should thus always be dumped on a separate file.}

Read the weight tables from a disk file. Both the $x$ and $\mu^2$ 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 $x$-$\mu^2$ 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

```fortran
  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, *nwtab )
```

Returns the size nwtot of the QCDNUM store (the parameter nwf0 in qcdnum.inc), the number of words used (nwuse) and the size of one pdf table (nwtab).

### 5.4 Parameters

In this section we describe the QCDNUM routines to set evolution parameters like the perturbative order, flavour thresholds, $\alpha_s$, etc. All these parameters have reasonable defaults but you can change them at any point in the code. Note that a re-definition of these evolution parameters invalidates the pdf tables of all existing types. The weight tables are not invalidated. In this way, all pdfs are always evolved with the same set of parameters; you cannot, for instance, have both un-polarised and polarised pdfs in memory, and evolve one in NNLO and the other in NLO.

```
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 $\alpha_s$ evolution the starting value $\text{alfs}$ and the starting renormalisation scale $r2$. Default $\alpha_s(m_Z^2) = 0.118$.

call SETCBT( nfix, iqc, iqk, iqt )

Select the FFNS or VFNS mode, and set thresholds on $\mu^2_F$.

$nfix$ Number of flavours in the FFNS mode. If not set to 3, 4, 5 or 6, QCDNUM runs in the VFNS mode.

$iqc, iqb, iqt$ 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 $iqt \geq iqk+2$.

A threshold index value of zero (or larger than the number of grid points) means ‘beyond the upper edge of the grid’. For instance, $(iqc,b,t) = (0,0,0)$ is like running in the FFNS with $n_f = 3$ 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 $\mu^2_R$.


$r2c, r2b, r2t$ Thresholds defined on the renormalisation scale $\mu^2_R$. When crossing a threshold, $n_f$ changes in the $\beta$-functions but not in the splitting functions.

To put a threshold below the evolution range, set it to a value $\leq 0$. 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
ncall 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.
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 \( \mu^2_F \) 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 \( \mu^2_R \) scale.

Define the relation between the factorisation scale \( \mu^2_F \) and the renormalisation scale \( \mu^2_R \)

\[
\mu^2_R = a_R \mu^2_F + b_R.
\]

Default: \( ar = 1 \) and \( br = 0 \).

Convert the factorisation scale \( \mu^2_F \) to the renormalisation scale \( \mu^2_R \) and vice versa.

Restrict the kinematic range of the pdf evolution to a part of the \( x-\mu^2 \) grid.

\( xmi, q2mi, q2ma \) Re-define the \( x-\mu^2 \) range.
\( dummy \) Not used at present.

The cuts will stay in effect until the next call to \texttt{setcut}; to release a cut you should set the corresponding value to \( \leq 0 \). We remind that \texttt{setcut} invalidates all the pdfs in memory so that it should always be followed by a call to \texttt{evolfg} or \texttt{pdfinp}.

QCDNUM treats a call to \texttt{setcut} as a \textit{request} and will, if necessary, open one or more cuts if they would lead to an empty kinematic domain.\(^{20}\)

A call to \texttt{getcut}—with the same argument list as \texttt{setcut}—returns the value of the current cuts. Note that the default cuts are set to the full range of the \( x-\mu^2 \) grid.

Restricting the evolution to the kinematic range of the data is a nice way to save CPU time in the \( \chi^2 \) 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.\(^{20}\)

\(^{20}\) Also the evolution routine \texttt{evolfg} may open the \( \mu^2 \)-range to ensure that \( \mu^0 \) falls inside. But this is only temporary and, on exit, the range will be closed again to what it was before the call to \texttt{evolfg}.\]
Lval = LPASSC( x, qmu2, *ifail, ichk )

The logical function \texttt{lpassc} returns \texttt{.true.} (\texttt{.false.}) if an \(x-\mu^2\) point passes (fails) the current cuts. Both \texttt{lval} and \texttt{lpassc} must be declared \texttt{logical} in the calling routine.

\begin{itemize}
  \item \texttt{x}, \texttt{qmu2} Input kinematic point.
  \item \texttt{ifail} Indicates which cut is failed (0 = all passed): 1 = \texttt{xmi}, 2 = \texttt{q2mi} or 3 = \texttt{q2ma}.
  \item \texttt{ichk} If set to 1, issue a fatal error message when \((x,\mu^2)\) does not pass the cuts.
\end{itemize}

Note, however, that all QCDNUM routines already handle points outside the kinematic range by producing either a null result or an error message. A call to \texttt{lpassc} is thus only useful if you want to catch out-of-range conditions before QCDNUM, and take some corrective action or print your own message. For this, you can customise the \texttt{lpassc} error message by an upstream call to \texttt{setUmsg} (see Section 6.7) which will define the subroutine name that will be printed in the message text.

```fortran
subroutine MySubroutine(x,qmu2)
  ...
  call setUmsg( 'MySubroutine ( x, qmu2 )' ) !define text
  ...
  ichk = 1 !print message if x or qmu2 out of range
  ...
  Lval = Lpassc(x,qmu2,ifail,ichk)
  ...
  call clrUmsg !always clear the text before return

5.5 Evolution

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

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

\begin{itemize}
  \item \texttt{r2} Renormalisation scale \(\mu_R^2\) where \(\alpha_s\) is to be calculated.
  \item \texttt{nf} Returns, on exit, the number of flavours at the scale \texttt{r2}.
  \item \texttt{ierr = 1} Too low value of \texttt{r2}. Internally, there is a cut \(r2 > 0.1\) GeV\(^2\) and also a cut on the slope, to avoid getting too close to \(\Lambda^2\).
\end{itemize}

The input scale and input value of \(\alpha_s\), the order of the evolution and the flavour thresholds are those set by default or by the routines described in Section 5.4. Note that although \(\alpha_s\) is evolved on the renormalisation scale the result, in the VFNS, may still depend on the relation between \(\mu_R^2\) and \(\mu_F^2\). This is because the position of the heavy flavour thresholds depends on this relation.\textsuperscript{21}

\textsuperscript{21}If \(\mu_F^2 = a \mu_R^2 + b\) then the flavour thresholds are similarly related: \(\mu_{h,F}^2 = a \mu_{h,R}^2 + b\). In this way, \(n_f\) changes simultaneously in both the splitting and the \(\beta\)-functions, as required (see Section 2.5).
Evolve a complete set of parton momentum densities from an input scale $\mu_0^2$. If QCDNUM runs in the FFNS, the gluon and $2n_f$ quark densities must be given as an input at $\mu_0^2$. In the VFNS, the gluon and $2n_f = 6$ light quark densities must be given at $\mu_0^2 < \mu_c^2$.

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\bar{t}$</th>
<th>$\bar{b}$</th>
<th>$\bar{c}$</th>
<th>$\bar{s}$</th>
<th>$\bar{u}$</th>
<th>$\bar{d}$</th>
<th>$g$</th>
<th>$d$</th>
<th>$u$</th>
<th>$s$</th>
<th>$c$</th>
<th>$b$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-6$</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

The input function `func` must be coded as follows

```c
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
```

The input function `func` must be coded as follows

```c
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
```

```
```
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:

```fortran
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.

```fortran
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
```

### 5.6 External Pdfs

In QCDNUM, you can read up to 5 different pdf sets from some external source, with type identifiers running from $id = 5$ to $9$. Care should be taken that the perturbative order, the flavour scheme, the positions of the thresholds and the input value of $\alpha_s$ are set correctly in QCDNUM because otherwise you will get the wrong answer when the pdf set is used later on in structure function or cross-section calculations. Because a downstream call to one of the routines in Section 5.4 would invalidate the pdfs, it is important that all evolution parameters are defined before the call to `pdfinp`.

```fortran
call PDFinp ( subr, iset, offset, *epsi, *nwds )
```

- **subr** User supplied subroutine (see below), declared `external` in the calling routine.
- **iset** Pdf set identifier in the range 5–9. If the pdf set exists, it will be overwritten.
- **offset** Relative offset $\delta$ at the thresholds $\mu^2_h$ which is used to catch discontinuities at the thresholds, if any, by sampling the pdfs at $\mu^2_h(1 \pm \delta)$. Usually you can set $\delta$ 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.
- **epsi** Maximum deviation of the quadratic spline interpolation from linear interpolation mid-between the grid points, as is described for the routine `evolfg`.
nwds Last word occupied in the store. Fatal error if the store is not large enough.

The routine `subr` provides the interface between `QCDNUM` and the external repository:

```fortran
subroutine SUBR ( x, qmu2, xpdf )
  implicit double precision (a-h,o-z)
  dimension xpdf(-6:6)
  xpdf(0) = xglue_from_elsewhere(x,qmu2)
..```

On exit, the array `xpdf(-6:6)` must contain the values of the gluon and the (anti-)quark momentum densities at $x$ and $\mu^2$, indexed according to (5.1); note the PDG convention.

### 5.7 Pdf Interpolation

Here we describe routines to access the gluon distribution ($xg$), the quark and anti-quark distributions ($xq, x\bar{q}$), or linear combinations of the quarks and anti-quarks. It is also possible to directly access the basis singlet/non-singlet pdfs in memory ($xe^\pm$, defined in Section 2.4). These routines perform local polynomial interpolation on a $k \times 3$ mesh around the interpolation point in $x$ and $\mu^2$, where $k$ is the current interpolation order in $x$. Two routines are provided to investigate the behaviour of the internal spline representation in $x$.

```
Lval = CHKPDF( iset )
```

Returns `.true.` if the pdf set [1–9] exists in memory. Both `Lval` and `chkpdf` should be declared `logical` in the calling routine.

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

Returns the gluon density or one of the (anti-)quark densities, interpolated to $x$ and $\mu^2$.

- **iset**: Pdf set identifier [1–9] indicating un-polarised (1), polarised (2), fragmentation function (3), custom (4), or external pdfs (5–9).
- **id**: Gluon, quark or anti-quark identifier, indexed as given in (5.1).
- **x, qmu2**: Input value of $x$ and $\mu^2$.
- **ichk**: If set to zero, `fvalxq` will return a null value when $x$ or `qmu2` are outside the grid boundaries or cuts; if set to a non-zero value a fatal error message will be issued.

```
call FPDFXQ ( iset, x, qmu2, *pdfs, ichk )
```

Returns all pdf values in one call. The arguments are as given above, except
pdfs  Output array, dimensioned to pdfs(-6:6) in the calling routine. The indexing is given in (5.1).

\[ \text{pdf} = \text{FSUMXQ} \left( \text{iset, def, x, qmu2, ichk} \right) \]

Return a weighted sum of quark densities. The arguments are as given above, except

def  Input array, dimensioned to def(-6:6) in the calling routine, containing the coefficients of the linear combination. The indexing is as given in (5.1) but note that def(0) is ignored since it does not correspond to a quark density.

\[ \text{pdf} = \text{FSNSXQ} \left( \text{iset, id, x, qmu2, ichk} \right) \]

Return the gluon density or one of the singlet/non-singlet basis pdfs. The arguments are as given above, except that id is now indexed as follows:

\[ \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
g & q & e_2^+ & e_3^+ & e_4^+ & e_5^+ & e_6^+ & q_v & e_2^- & e_3^- & e_4^- & e_5^- & e_6^- \\
\end{array} \]  

(5.2)

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.

\[ \begin{align*}
\text{fvalij} & \left( \text{iset, id, ix, iq, ichk} \right) \\
\text{fpdfij} & \left( \text{iset, ix, iq, pdfs, ichk} \right) \\
\text{fsumij} & \left( \text{iset, def, ix, iq, ichk} \right) \\
\text{fsnsij} & \left( \text{iset, id, ix, iq, ichk} \right) \\
\end{align*} \]

These routines are of course faster since no interpolation is required.

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

\[ \text{epsi} = \text{SPLCHK} \left( \text{iset, id, iq} \right) \]

Returns \( \epsilon = \|u - v\| \) at a grid point iq. Here \( u \) and \( v \) are the vectors of quadratic and linear interpolation mid-between the grid points in x, as is described in Section 3.3. 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.

\[ \text{pdf} = \text{FSPLNE} \left( \text{iset, id, x, iq} \right) \]

This routine is identical to fsnsxq, 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 \( \mu^2 \)). 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 fsnsxq. You do not need fsplne or splchk to detect spline oscillations, since that is done automatically by evolfg and pdfinp.

43
5.8 Fast Interpolation

It may happen that you want to generate a large table of interpolated pdfs, to be used by an application outside QCDNUM for instance. Here are two fast routines to do this.

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

Process a list of interpolation points.

iset  
Pdf set identifier [1–9].
def  
Input array, dimensioned to def(−6:6) in the calling routine, containing the coefficients of a linear combination. The indexing is as given in (5.1). Set def(0) = 0 to obtain a linear combination of quarks. To interpolate the gluon distribution, set all coefficients to zero, except def(0).
x, qmu2  
Interpolation points, dimensioned to x(n) and qmu2(n) in the calling routine.
pdf  
Output list of pdf values, dimensioned to pdf(n) in the calling routine.
n  
Number of items in x, qmu2 and pdf.
ichk  
If non-zero, QCDNUM insists that all interpolation points are within the grid or cuts. If zero, a null value is returned for pdfs outside the boundaries.

```
call PDFTAB ( iset, def, x, nx, q, nq, *table, ichk )
```

Generate a pdf table in \( x \) and \( \mu^2 \). The parameters are as for `pdflst`, except,

x, nx  
\( x \)-grid, dimensioned to 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 \( \mu^2 \)-grid.
table  
Output table of pdfs, dimensioned to table(nx,nq) in the calling routine.

This routine runs a factor of two faster than `pdflst` because of simpler bookkeeping.

6 Convolution Engine

The QCDNUM convolution engine provides tools to calculate structure functions in deep inelastic scattering, hadron-hadron scattering cross-sections and parton luminosities. The engine drives the add-on package ZMSTF that computes the zero-mass structure functions \( F_2, F_L \) and \( xF_3 \) in un-polarised deep inelastic scattering. It is also used in the HQSTF package that computes the heavy flavour contributions to \( F_2 \) and \( F_L \) in the fixed flavour number scheme [16]. Both these packages are included in the QCDNUM distribution and are described in the Sections C.3 and D.2 of this write-up.

From the parton number densities \( f \) and kernels \( K \), all kind of convolution integrals can be calculated with the engine, such as

\[
x[f \otimes K](x), \quad x[f \otimes K_a \otimes K_b](x), \quad x[f_a \otimes f_b](x), \quad x[f_a \otimes f_b \otimes K](x), \quad \text{etc.}
\]
Here $\otimes$ stands for Mellin convolution as defined by (2.5). We refer to Section 3.2 for how convolution integrals are computed and where the factor $x$ in front comes from. We emphasise that the kernel $K$ must be defined by convolution with a number density. If not, then it must be transformed as necessary, before it is fed into QCDNUM.

The steps to be taken in a calculation based on the convolution engine are the following.

1. Declare one or more stores and partition these into tables. Then fill the tables with weights for all the convolution kernels needed in the calculation (Section 6.2);

2. Write a function $\text{myfun}(ix, iq)$ that returns the structure function, cross section or luminosity at a grid point in $x$ and $\mu^2$ (Section 6.3);

3. Pass $\text{myfun}$ to a QCDNUM routine that will take care of the interpolation to any desired $x$ and $\mu^2$ (Section 6.4).

This procedure is fairly straight-forward and therefore suitable for prototyping and debugging. However, there is a considerable amount of overhead so that it is recommended to ultimately move steps 2 and 3 of the computation to a fast calculation scheme that is described in Section 6.5. By this you will gain at least an order of magnitude in speed.

Before we present the convolution engine we will first, in the next section, introduce the rescaling variable $\chi$ to accommodate generalised mass variable flavour number schemes (GM-VFNS, see [28] for a recent review) in structure function calculations.

### 6.1 Rescaling Variable in Convolution Integrals

The general expression for a structure function can be written as

$$ F_i(x, Q^2) = \sum_j x \int_\chi^1 \frac{dz}{z} f_j(z, \mu^2) C_{ij} \left[ \frac{\chi}{z}, \mu^2, Q^2, m^2_h, \alpha_s(\mu^2) \right]. $$

(6.1)

Here the index $i$ labels the structure function (e.g. $F_2$, $F_L$, $xF_3$, $F^c_2$, ...) 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 $\mu^2$ and $Q^2$, on one or more quark masses $m^2_h$ and on the strong coupling constant $\alpha_s$. 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,

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

(6.2)

We have $0 \leq \chi \leq 1$ so that the range of $x$ in (6.1) is restricted to $0 \leq x \leq 1/a$. In the zero-mass limit $a = 1$, $\chi = x$, and (6.1) 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 $\alpha_s$ and the indices $i, j$)

$$ F(x, Q^2) = x \int_\chi^1 \frac{dz}{z} f(z, \mu^2) C \left( \frac{\chi}{z}, \mu^2, Q^2, m^2_h \right). $$

(6.3)
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 \( \chi \) is proportional to \( x \), (6.3) can be written as a weighted sum of spline coefficients

\[
\mathcal{F}(y, Q^2) = \sum_{j=1}^{i} W_{ij} A_j
\]

with \( W_{ij} = w_{i-j+1} \) and

\[
w_\ell = e^{-b} \int_{0}^{y_\ell - b} dz \ Y_1(z) \ D(y_\ell - b - z, t, Q^2, m_H^2) \quad (1 \leq \ell \leq n).
\]

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 (6.5) 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.

We emphasise that convolution integrals found in the literature must, if necessary, be brought into the general form (6.1) by modifying the published Wilson coefficient. An example of such a modification can be found in Appendix D.1.

### 6.2 Weight Tables

In this section we describe routines that partition a linear store into tables and fill these tables with weights used in the calculation of convolution integrals. It is important to realise that the convolution kernels may contain singularities, see also Appendix A. 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

\[
C(x) = A(x) + [B(x)]_+ + R(x)[S(x)]_+ + D(x)\delta(1-x).
\]

QCDNUM provides routines that can calculate weights for each term separately (if present) and add these to the weight table of \( C \).

For reasons of efficiency and economy of storage, there are four different types of tables:

- \( \text{itype} = 1 \) Weights that depend only on \( x \). Table identifiers run from 101–199;
- \( \text{itype} = 2 \) Weights that depend on \( x \) and \( n_f \). Identifiers run from 201–299;
- \( \text{itype} = 3 \) Weights that depend on \( x \) and \( \mu^2 \). Identifiers run from 301–399;
- \( \text{itype} = 4 \) Weights that depend on \( x \), \( \mu^2 \) and \( n_f \). Identifiers run from 401–499.

Although it is a good idea to take out as many \( \mu^2 \)-dependent factors as possible from the convolution kernel, it is clear from (6.3) that quark mass parameters and the relation between \( \mu^2 \) and \( Q^2 \) may enter via the rescaling variable \( \chi \) and that this dependence can never be factored out of the convolution integral. Thus the weight tables of the GM schemes will, in general, depend on \( x \) and \( \mu^2 \) and must be stored in type-3 or 4 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.

In Table 3 we list all available weight routines.

### Table 3: QCDNUM convolution weight table routines.

<table>
<thead>
<tr>
<th>Subroutine or function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOKTAB (w, nw, itypes, *nwords)</td>
<td>Partition into tables</td>
</tr>
<tr>
<td>MAKEWTA (w, id, afun, achi)</td>
<td>Regular piece $A(x)$</td>
</tr>
<tr>
<td>MAKEWTB (w, id, bfun, achi, nodelta)</td>
<td>Singular piece $[B(x)]_+$</td>
</tr>
<tr>
<td>MAKEWRS (w, id, rfun, sfun, achi, nodelta)</td>
<td>Product $R(x)[S(x)]_+$</td>
</tr>
<tr>
<td>MAKEWTD (w, id, dfun, achi)</td>
<td>Delta function $D(x)\delta(1-x)$</td>
</tr>
<tr>
<td>MAKEWTX (w, id)</td>
<td>Weight table for $x[f_a \otimes f_b]$</td>
</tr>
<tr>
<td>SCALEWT (w, c, id)</td>
<td>Scale weight table</td>
</tr>
<tr>
<td>IDSPFUN ('pij', iord, itype)</td>
<td>Splitting function index</td>
</tr>
<tr>
<td>COPYWGT (w, id1, id2, iadd)</td>
<td>Copy weight table</td>
</tr>
<tr>
<td>WCRossW (w, ida, idb, idc, iadd)</td>
<td>Double convolution weights</td>
</tr>
<tr>
<td>WTIMESF (w, fun, id1, id2, iadd)</td>
<td>Multiply by $f(\mu^2, n_f)$</td>
</tr>
<tr>
<td>SETWPAR (w, pars, n)</td>
<td>Store extra information</td>
</tr>
<tr>
<td>GETWPAR (w, *pars, n)</td>
<td>Read extra information</td>
</tr>
<tr>
<td>TABDUMP (w, lun, 'filename', 'key')</td>
<td>Dump to disk</td>
</tr>
<tr>
<td>TABREAD (w, n, lun, 'fn', 'key', *nw, *ierr)</td>
<td>Read from disk</td>
</tr>
</tbody>
</table>

Output arguments are pre-fixed with an asterisk (*).

```call BOOKTAB (w, nw, itypes, *nwords)```

Partition a store `w` into tables.

- **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(4)` 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.
- **nwords**: Gives, on exit, the number of words used in the store. If `nwords` is negative, then the store is not sufficiently large and should be re-dimensioned in the calling routine to at least `-nwords`.

Note that you can declare and partition as many stores as desired, one per structure function for instance. Note also that `booktab` initialises all tables in the store to zero.\(^{22}\)

\(^{22}\)You can use the routine `scalewt` to explicitly zero a table, if you want to.
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 store $w$.

$w$ Store declared in the calling routine and previously partitioned by booktab.
$id$ Table identifier. To add results to a type-$n$ table, you should use identifiers in the range n01–n99, with $n = 1, 2, 3$ or $4$.
$afun$ User function (see below) returning the regular piece of the convolution kernel. Should be declared external in the calling routine.
$achi$ User function (see below), declared external in the calling routine, that returns the value $a$ of the rescaling variable $\chi = ax$.

The function $afun$ provides an interface between QCDNUM and the regular part of the kernel $C(\chi, \mu^2, Q^2, m_h^2)$ and should be coded as follows.\(^23\)

```fortran
double precision function afun(chi,qmu2,nf)  !chi = a*x
implicit double precision (a-h,o-z)
common /fixpar/ par1, par2, .....  !parameters, if any
Q2 = some_function_of(qmu2,some_params)  !Q2
afun = cfun(chi,qmu2,Q2,nf,some_params)  !convolution kernel
return
end
```

The function $achi$ should return, as a function of $\mu^2$, the factor $a$ that defines the rescaling variable $\chi = ax$.

```fortran
double precision function achi(qmu2)
implicit double precision (a-h,o-z)
common /fixpar/ par1, par2, .....  !parameters, if any
Q2 = some_function_of(qmu2,some_params)  !Q2
achi = some_function_of(Q2,some_params)
return
end
```

QCDNUM insists that always $achi \geq 1$, you will get a fatal error if not. To compute standard Mellin convolutions $x[f \otimes C](x)$, simply set $achi = 1$ for all $\mu^2$.

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

\(^23\)We assume here that the kernel conforms to (6.1). If not, then $afun$ must take care of this.
Calculate the weights for the singular contribution $[B(x)]_+$ to a convolution kernel and add these to a table in the store $w$. 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

```fortran
    call makewta(w,201,afun,achi)  !put weights in id = 201
    call makewtb(w,201,bfun,achi,0) !add weights to id = 201
```

It is seen from Appendix A, equation (A.4), 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.

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

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

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

- **w**: Store declared in the calling routine and previously partitioned by `booktab`.
- **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.

Multiply the contents of table $id$ by a constant $c$.

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

Return the index ($< 0$) of a splitting function weight table stored internally in QCDNUM.

- **'pij'**: Name of the splitting function. Valid input strings are

  ```
  PQQ, PQG, PGQ, PGG, PPL, PMI, PVA.
  ```
iord  Select LO (1), NLO (2) or NNLO (3).
itype  Select evolution type: un-polarised (1), polarised (2), fragmentation function (3) or custom (4).

The index returned by idspfun is encoded as -(1000*iday+id), where id is the internal table identifier. If the table does not exist, the function returns a value of -1.

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

Copy the contents of table id1 to id2.

w    Store declared in the calling routine.
id1   Input table identifier. You can copy a splitting function weight table from internal QCDNUM memory to the store by setting id1 < 0. Valid identifiers are generated by idspfun(), as described above.
id2   Output table identifier with id2 \neq id1. The output table type may be different from the input table type, see below.
iadd  If set to 0 copy id1 to id2, if set to +1 (-1) add (subtract) id1 to (from) id2.

For this routine—and for those described below—the output table type can be different from the input table type, provided that this does not lead to a loss of input information. Thus you can copy a type-1 table to a type-3 table but not the other way around (fatal error). To avoid errors it is best to copy a splitting function weight table to type-2.

```
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 $W_a$ and $W_b$.

w    Store declared in the calling routine.
ida   Table identifier containing the weights of kernel $K_a$. When ida < 0 you will access a splitting function weight table which is stored internally in QCDNUM. See idspfun() above for how to generate a valid splitting function identifier.
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.

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 ($x, \mu^2$), then idc must be type-4 ($x, \mu^2, n_f$). The routine checks this.

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

Multiply a weight table by a function of $\mu^2$ and $n_f$ and store the result in another table.
Store declared in the calling routine.

User supplied double precision function fun(iq,nf) declared external in the calling routine.

Input weight table identifier. It is possible to access a splitting function weight table by setting id1 < 0. Valid identifiers can be obtained from idspfun() described above.

Identifier of the output table. It is allowed to have id1 = id2 (in-place modification of a table), unless id1 is a splitting function table. The table type of id2 must be such that no information is lost. The routine checks this.

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:

<table>
<thead>
<tr>
<th>type</th>
<th>variables</th>
<th>iq range</th>
<th>nf range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1–1</td>
<td>3–3</td>
</tr>
<tr>
<td>2</td>
<td>x, nf</td>
<td>1–1</td>
<td>3–6</td>
</tr>
<tr>
<td>3</td>
<td>x, \mu^2</td>
<td>1–nq</td>
<td>3–3</td>
</tr>
<tr>
<td>4</td>
<td>x, \mu^2, nf</td>
<td>1–nq</td>
<td>3–6</td>
</tr>
</tbody>
</table>

With this routine you can, in combination with wcrossw, construct weight tables for combinations of convolution kernels, such as those given in (C.7). For instance, here is code that generates a table for

\[ C^{(2,1)}_{2,+} = C^{(0)}_{2,q} \otimes P^{(1)}_{+} + C^{(1)}_{2,+} \otimes P^{(0)}_{qq} - \beta_0 C^{(1)}_{2,+}. \]

!beta function

```fortran
external beta0
```

```fortran
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 )
```

Write extra information to the store, for instance quark masses or other parameters that you may want to dump to disk, together with the tables themselves.

Store, partitioned by a previous call to booktab.

List of parameters to be written. Should be dimensioned to at least par(n) in the calling routine.

Number of items to be written up to a maximum of miw0 = 20. If necessary, you can change the value of miw0 in qcdnum.inc and recompile QCDNUM.

The parameters can be read back by a call to getwpar(w,par,n).

51
Dump the store \( w \) to disk. Apart from the store, information is written about the QCDNUM version, the \( x-\mu^2 \) grid definition and the current spline interpolation order. The \texttt{key} text string can be used to stamp the file with a version number or other identifier. The dump is unformatted so that the file cannot be exchanged across machines.

Read a store from disk into the array \( w(nw) \). The size of the store (in words) is returned in \( nwords \). You will get a fatal error message if \( w(nw) \) is not large enough to contain the store. Note that the \( x \) and \( \mu^2 \) grids must have been defined before the call to \texttt{tabread}.

On exit, the error flag is set as follows (non-zero means that nothing has been read in).

\begin{itemize}
  \item 0 Store successfully read in.
  \item 1 Read error or input file does not exist.
  \item 2 File written by another QCDNUM version.
  \item 3 Key mismatch.
  \item 4 Incompatible \( x-\mu^2 \) grid definition.
\end{itemize}

QCDNUM insists that the key written on the file matches the key entered as an argument to \texttt{tabread}.\footnote{Note that the key matching is case insensitive and that leading and trailing blanks are ignored.} 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 \texttt{tabdump} and \texttt{tabread}.

### 6.3 Convolution

In Table 4 we list the routines that can be used to build a structure function, cross-

<table>
<thead>
<tr>
<th>Subroutine or function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCROSSK ( ( w, idw, iset, idf, ix, iq ) )</td>
<td>Convolution ( x[f \otimes K] )</td>
</tr>
<tr>
<td>FCROSSF ( ( w, idw, iset, ida, idb, ix, iq ) )</td>
<td>Convolution ( x[f_a \otimes f_b] )</td>
</tr>
<tr>
<td>EFROMQQ ( ( qvec, *evec, nf ) )</td>
<td>Transform from ( q, \bar{q} ) to ( e^\pm )</td>
</tr>
<tr>
<td>QQFROME ( ( evec, *qvec, nf ) )</td>
<td>Transform from ( e^\pm ) to ( q, \bar{q} )</td>
</tr>
<tr>
<td>NFLAVOR ( ( iq ) )</td>
<td>Returns ( n_f )</td>
</tr>
<tr>
<td>GETALFN ( ( iq, n, *ierr ) )</td>
<td>Returns ( (\alpha_s/2\pi)^n )</td>
</tr>
</tbody>
</table>

Output arguments are pre-fixed with an asterisk (*).

section or parton luminosity at a grid point in \( x \) and \( \mu^2 \).
A convolution is always computed with the pdfs in QCDNUM memory, that is, with the gluon density or with one of the singlet/non-singlet quark densities $|e^{\pm}\rangle$ as defined in Section 2.4. To translate a linear combination of quarks and anti-quarks to the $|e^{\pm}\rangle$ basis, and vice versa, the routines efrommq and qfrome are provided.

For convenience we show here again the indexing (5.2) of the singlet/non-singlet basis

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
g & q_s & e^+_2 & e^-_3 & e^+_4 & e^-_5 & q_v & e^-_2 & e^+_3 & e^-_4 & e^+_5 & e^-_6 & e^-_6 \\
\end{array}
\]  
(6.7)

and the indexing (5.1) of the flavour basis

\[
\begin{array}{cccccccccccc}
-6 & -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
i & b & \bar{c} & \bar{s} & \bar{u} & \bar{d} & g & d & u & s & c & b & t \\
\end{array}
\]  
(6.8)

\[\text{val} = \text{FCROSSK} \left( w, \text{idw}, \text{iset}, \text{idf}, \text{ix}, \text{iq} \right)\]

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

- **w**: Store declared in the calling routine and previously filled with weights.
- **idw**: Identifier of a table in the store $w$.
- **iset**: Pdf set identifier [1–9].
- **idf**: Pdf identifier [0–12], indexed according to (6.7).
- **ix, iq**: Indices of an $x$-$\mu^2$ grid point.

Splitting function tables cannot be directly accessed by this routine; they should first be copied to the store by a call to copywgt.

\[\text{val} = \text{FCROSSF} \left( w, \text{idw}, \text{iset}, \text{ida}, \text{idb}, \text{ix}, \text{iq} \right)\]

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

- **w**: Store declared in the calling routine and previously partitioned by booktab.
- **idw**: Identifier of a weight table, previously filled by a call to makewx.
- **iset**: Pdf set identifier [1–9].
- **ida, idb**: Pdf identifiers [0–12], indexed according to (6.7).
- **ix, iq**: Indices of an $x$-$\mu^2$ grid point.

To convolute two linear combinations of pdfs, you have to multiply these out, compute each term with fcrossf, and then add-up the results. Note that such a convolution is much easier calculated with the fast routines described in Section 6.5.

\footnote{Un-polarised (1), polarised (2), fragmentation function (3), custom (4), or external (5–9).}
Transform the coefficients of a linear combination of quarks and anti-quarks from the flavour basis to the singlet/non-singlet basis as described in Section 2.4.

\( \text{call EFROMQQ(} \ qvec, *evec, nf \ \text{)} \)

\( qvec \) Input array, dimensioned \( qvec(-6:6) \), filled with the coefficients of a linear combination of quarks and anti-quarks and indexed according to (6.8).

\( evec \) Output array, dimensioned to \( evec(12) \), filled with the coefficients written on the singlet/non-singlet basis, indexed according to (6.7).

\( nf \) Active number of flavours. This parameter is needed to construct the appropriate \( 2n_f \times 2n_f \) transformation matrix that acts on \( qvec(-nf:nf) \).

Thus if a linear combination of quarks and anti-quarks is written as

\[
|p\rangle = \sum_{i=1}^{n_f} (\alpha_i|q_i\rangle + \beta_i|\bar{q}_i\rangle) = \sum_{i=1}^{n_f} (d_i^+|e_i^+\rangle + d_i^-|e_i^-\rangle) \tag{6.9}
\]

and \( \alpha_i \) and \( \beta_i \) are stored in the input vector \( qvec \), then the coefficients \( d_i^\pm \) are returned in the output vector \( evec \).

\( \text{call QQFROME(} \ evec, *qvec, nf \ \text{)} \)

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 \( \text{efromqq} \).

\( \text{nf = NFLAVOR( } \ iq \ \text{)} \)

Returns the number of active flavours at the grid point \( iq \). Note that this number is \((4,5,6)\) and not \((3,4,5)\) at the thresholds \((iqc,iqb,iqt)\).

\( \text{as = GETALFN( } \ iq, n, *ierr \ \text{)} \)

Returns the value of \((\alpha_s/2\pi)^n\) at the factorisation scale \( \mu_F^2 \). Here \( \alpha_s \) is computed from the Taylor expansion (2.17), appropriately truncated depending on the current value of the perturbative order \( iord \). Because the truncation is different for the pdfs (Section 2.3) and the structure functions (Appendix C.2), the value of \( n \) must be set as follows.

- If the convolution at (LO, NLO, NNLO) should be multiplied by \((\alpha_s, \alpha_s^2, \alpha_s^3)\) then you set \( n = (1,2,3) \) in the call to \( \text{getalfn} \).
- If the convolution at (LO, NLO, NNLO) should be multiplied by \((1, \alpha_s, \alpha_s^2)\) then you set \( n = (0,-1,-2) \) in the call to \( \text{getalfn} \).
- If \( n > iord \), the value of \((\alpha_s/2\pi)^n\) is calculated at \( \mu_R^2 \), instead of at \( \mu_F^2 \). (This is already the case for \( n = iord \), see Section 2.3.)
To have access to the NNLO discontinuities at the thresholds, you can set \( iq \) positive (includes discontinuity) or negative (does not include discontinuity), thus:

\[
\begin{align*}
call \text{getalfn} (iq\text{charm}, n, ierr) & \quad ! \text{result for } nf = 4 \\
call \text{getalfn} (-iq\text{charm}, n, ierr) & \quad ! \text{result for } nf = 3
\end{align*}
\]

In other words, by preceding \( iq \) with a minus sign you effectively change the QCDNUM default \( n_f = (4, 5, 6) \) at the thresholds to the alternative \( n_f = (3, 4, 5) \). When \( iq \) is close to or below the value of \( \Lambda^2 \), then \( ierr = 1 \) and \( \text{getalfn} \) returns the null value. This also happens when \( iq \) is outside the grid boundaries (\( ierr = 2 \)).

Note that the QCDNUM expansion parameter is \( \alpha_s/2\pi \) but that many convolution kernels found in the literature are defined for an expansion in \( \alpha_s/4\pi \), in which case you must account for the appropriate factors of 2 somewhere in the calculation.

### 6.4 Interpolation

With the routines presented above you can write a function \( \text{stfun}(ix,iq) \) that returns the value of a structure function or cross section at a grid point in \( x \) and \( \mu^2 \). The routine \( \text{stfunxq} \) then takes care of the interpolation to any value of \( x \) and \( \mu^2 \). This interpolation is done on a \( k \times 3 \) mesh around the interpolation point, where \( k \) is set to the current spline interpolation order (\( 2 = \text{linear}, 3 = \text{quadratic} \)). Thus \( 3k \) functions have to be computed for each interpolation which becomes inefficient if there are interpolations with overlapping meshes. By processing lists of interpolation points, instead of each point individually, redundant calculations are avoided which can lead to considerable gains in computing time. In other words, \( \text{stfunxq} \) should not be called in a loop over interpolation points but be given the list of points.

\[
\begin{align*}
call \text{STFUNXQ} ( \text{stfun}, x, qmu2, stf, n, ichk )
\end{align*}
\]

Interpolate the function \( \text{stfun}(ix,iq) \) to a list of \( x \) and \( \mu^2 \) values.

- **stfun**: Double precision function, declared external in the calling routine, that returns the value of a structure function or cross-section at \( (ix,iq) \).
- **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 \( \mu^2 \) 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 \( \mu^2 \) and not in \( Q^2 \).
6.5 Fast Computation

As already remarked above, the routines provided up to now are fine for prototyping but are slow because there is quite a lot of overhead when the calculation is repeated at more than one interpolation point. Here we describe a set of routines that does optimised bulk calculations on selected points in the \(x-\mu^2\) grid. With these fast routines you can easily gain one or two orders of magnitude in speed. To make the calculation flexible it is broken down into small steps where intermediate results are stored into scratch buffers (by default, the fast engine generates 5 scratch buffers but you can have more, if necessary, see below). An optimised calculation proceeds as follows.

1. Pass a list of interpolation points in \(x\) and \(\mu^2\) to a QCDNUM routine that determines which grid points will be occupied in the course of the calculation;

2. Store a pdf or a linear combination of pdfs in a scratch buffer;

3. Convolute the pdf with a convolution kernel or a perturbative expansion of kernels;

4. Multiply the convolution by a function of \(x\) and \(\mu^2\), for instance by some power of \(\alpha_s\) or by some kinematic factor;

5. Accumulate the cross section or structure function in a final buffer;

6. Pass this buffer to an interpolation routine to get a list of interpolated results.

The list of subroutines is given in Table 5. 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 \textit{sparse}. A convolution routine, on the other hand, does not only need the values

<table>
<thead>
<tr>
<th>Subroutine or function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FASTINI (x, qmu2, n, ichk)</td>
<td>Pass list of (x) and (\mu^2) values</td>
</tr>
<tr>
<td>FASTCLR (id)</td>
<td>Clear buffer</td>
</tr>
<tr>
<td>FASTEPM (iset, idf, idout)</td>
<td>Store (</td>
</tr>
<tr>
<td>FASTSNS (iset, pdf, isel, idout)</td>
<td>Store singlet/non-singlet component</td>
</tr>
<tr>
<td>FASTSUM (iset, coef, idout)</td>
<td>Store weighted sum of (</td>
</tr>
<tr>
<td>FASTFXK (w, idw, idf, idout)</td>
<td>Convolution (xf \otimes K(x))</td>
</tr>
<tr>
<td>FASTFXF (w, idw, ida, idb, idout)</td>
<td>Convolution (xf_a \otimes f_b(x))</td>
</tr>
<tr>
<td>FASTKIN (id, fun)</td>
<td>Scale by a kinematic factor</td>
</tr>
<tr>
<td>FASTCPY (idin, idout, iadd)</td>
<td>Copy or accumulate result</td>
</tr>
<tr>
<td>FASTFXQ (id, *f, n)</td>
<td>Interpolation</td>
</tr>
</tbody>
</table>

Output arguments are pre-fixed with an asterisk (*).
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 on 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.

The pdf set parameter \( \text{iset} \) in the routines \texttt{fastepm, fastsns} and \texttt{fastsum} selects the pdf set, namely, un-polarised (1), polarised (2), fragmentation function (3), custom (4), or external (5–9).

```fortran
call FASTINI ( x, qmu2, n, ichk )
```

Pass a list of interpolation points and, at the first call, generate the set of scratch buffers.

\( x \quad \text{Array, dimensioned to at least} \ n \ \text{in the calling routine, filled with} \ x \ \text{values.} \)

\( qmu2 \quad \text{As above, but for} \ \mu^2 \ (\text{not} \ Q^2). \)

\( n \quad \text{Number of entries in} \ x \ \text{and} \ qmu2. \)

\( ichk \quad \text{If non-zero, fastini insists that all} \ x \ \text{and} \ \mu^2 \ \text{are within the grid boundaries.} \)

By default, 5 scratch buffers (\( \text{id} = 1–5 \)) are generated at the first call (or cleared if they exist). This number can be changed by calling \texttt{setint(‘ntab’,ival)} prior to \texttt{fastini}. You will get a fatal error if there is not enough space for the scratch buffers, in which case you have to increase the value of \texttt{nwf0} in \texttt{qcdnum.inc}, and recompile QCDNUM. The number of interpolations is restricted to \( n < \texttt{mpt0} = 5000 \). If you want more interpolations in one call, you can increase the value of \texttt{mpt0} in \texttt{qcdnum.inc}, and recompile QCDNUM. However, large values of \texttt{mpt0} are not recommended because the bookkeeping space becomes very large: it is then better to split the calculation into chunks of 5000 interpolations, and proceed in batches.\(^{26}\)

```fortran
call FASTCLR ( \text{id} )
```

Clear a scratch buffer. Setting \text{id} = 0 will clear all buffers.

```fortran
call FASTEPM ( \text{iset, idf, idout} )
```

Copy the gluon density or one of the basis pdfs \( |e^\pm\rangle \) to a scratch table.

\( \text{iset} \quad \text{Input pdf set identifier [1–9].} \)

\( \text{idf} \quad \text{Pdf identifier [0–12], indexed according to (6.7).} \)

\( \text{idout} \quad \text{Output scratch table identifier [1–5].} \)

By default, \texttt{fastepm} generates a dense buffer; a sparse buffer is generated when you pre-pend the output identifier with a minus sign.

\(^{26}\)The example program \texttt{longlist.f} shows how you can use buffering to avoid the \texttt{mpt0} limit.
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  Input pdf set identifier [1–9].
df  Input array, dimensioned pdf(-6:6), filled with the coefficients of a linear combination of quarks and anti-quarks and indexed according to (6.8).
isel  Selection flag [0–7], see below.
idout  Output scratch table identifier [1–5].

The isel flag selects the (weighted) gluon distribution (0), the singlet component $q_s$ (1), the non-singlet component $q_{ns}^+$ (2), the valence component $q_v$ (3), the non-singlet component $q_{ns}^-$ (4), the sum $q_v + q_{ns}^-$ (5), all non-singlets $q_v + q_{ns}^- + q_{ns}^+$ (6) or all quarks (7).

By default, fastsns generates a dense buffer; a sparse buffer is generated when you pre-pend the output identifier with a minus sign.

Copy a linear combination of basis pdfs $|e^\pm\rangle$ to a scratch table.

iset  Input pdf set identifier [1–9].
coef  Array of coefficients dimensioned coef(0:12,3:6) in the calling routine.
idout  Output scratch table identifier [1–5].

The array coef(i,nf) is indexed according to (6.7). Here is code that fills coef by transforming a set of quark coefficients from flavour space to singlet/non-singlet space:

```fortran
  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 pre-pend the output identifier with a minus sign.
call FASTFXK ( w, idw, idf, idout )

Calculate the convolution $x[f \otimes K](x)$ at all selected grid points.

w Store, declared in the calling routine and previously filled with weights.
idw Set of weight identifiers, declared idw(4) in the calling routine, see below.
idf Input scratch buffer, previously filled by fastepm, fastsns or fastsum.
idout Output scratch table with idout $\neq$ idf.

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:

1. To convolute with a given weight table, set idw(1) to the identifier of that weight table and set idw(2), idw(3) and idw(4) to zero;

2. To convolute with a perturbative expansion, 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, as is the case for $F_1$ at LO, for instance. Declare in idw(4) the leading power of $\alpha_s$, that is, multiply (LO,NLO,NNLO) by $(1, \alpha_s, \alpha_s^2)$ if idw(4) = 0 and by $(\alpha_s, \alpha_s^2, \alpha_s^3)$ if idw(4) = 1. Note that the perturbative expansion is summed up to the current perturbative order, as set by default (NLO) or by an upstream call to setord. So if you run in LO you need only to fill idw(1) and idw(4).

The routine only accepts a dense buffer as input (otherwise fatal error) and will, by default, generate a sparse buffer as output. If you pre-pend the output identifier with a minus sign, the output buffer will be dense. In this way, the output table can serve as an input to another convolution which allows you to calculate multiple convolutions in a chain. For example,\footnote{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 $K_3$ with $f$.}

```
call fastSum( 1, coef, 1 ) ! 1 = f
call fastFxK( w, idK1, 1, -2 ) ! 2 = f * K1
call fastFxK( w, idK2, 2, 3 ) ! 3 = f * K1 * K2
```

call FASTFXF ( w, idx, ida, idb, idout )

Calculate the convolution $x[f_a \otimes f_b](x)$ at all selected grid points.

w Store, declared in the calling routine and previously partitioned by booktab.
idx Identifier of a weight table, previously filled by a call to makevtex.
ida, idb Identifiers of input scratch tables. It is allowed to have ida = idb.
idout Output scratch table with idout $\neq$ ida or idb.
As above, the routine accepts only dense buffers as input, and generates a sparse buffer as output, unless the output identifier is pre-pended by a minus sign, thus,

\[
\begin{align*}
\text{call fastSum( 1, coefa, 1 )} & \quad ! 1 = fa \\
\text{call fastSum( 1, coefb, 2 )} & \quad ! 2 = fb \\
\text{call fastFxF( w, idwX, 1, 2, -3 )} & \quad ! 3 = fa \ast fb \\
\text{call fastFxK( w, idwK, 3, 4 )} & \quad ! 4 = fa \ast fb \ast K \\
\end{align*}
\]

call FASTKIN ( id, fun )

Multiply the contents of a scratch table by a kinematic factor.

<table>
<thead>
<tr>
<th>id</th>
<th>Identifier of the input scratch table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun</td>
<td>Double precision function, declared \textbf{external} in the calling routine.</td>
</tr>
</tbody>
</table>

The routine loops over the selected grid points and calls the user supplied function \textit{fun} that should return the kinematic factor. The syntax of \textit{fun} is

\[
\text{double precision function } \textit{fun} ( \text{ix, iq, nf, ithresh} )
\]

\[
\begin{align*}
\text{ix, iq} & \quad \text{Grid point indices.} \\
\text{nf} & \quad \text{Number of flavors at iq. This number is bi-valued at the thresholds so that at the charm threshold, for instance, nf can be either 3 or 4.}^{28} \\
\text{ithresh} & \quad \text{Set to 0 if iq is not at a threshold and to } +1 (-1) \text{ if iq is at a threshold with the upper (lower) number of flavours. This variable can be used to take NNLO discontinuities into account, as is shown in the example below.}
\end{align*}
\]

\[
\begin{align*}
\text{double precision function } \textit{fkin(ix,iq,nf,ithresh)} \\
& \quad \ldots \\
& \quad \text{if(ithresh.ge.0) then} \\
& \quad \quad \text{alfas = getalfn( iq,1,ierr) } \quad !\text{alfas/2pi with discontinuity} \\
& \quad \text{else} \\
& \quad \quad \text{alfas = getalfn(-iq,1,ierr) } \quad !\text{without discontinuity} \\
& \quad \text{endif} \\
& \quad \ldots \\
\end{align*}
\]

call FASTCPY ( idin, idout, iadd )

Copy or accumulate a result in an output buffer.

<table>
<thead>
<tr>
<th>idin</th>
<th>Identifier of the input scratch table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>idout</td>
<td>Identifier of the output scratch table with \text{idout} \neq \text{idin}.</td>
</tr>
</tbody>
</table>

\[^{28}\text{The reader may wonder when QCNUM returns the value 3, and when the value 4. This depends on the interpolation point } \mu^2 \text{ to which iq is associated: if } \mu^2 \text{ is below (above) } \mu_c^2, \text{ then nf } = 3 (4).\]
iadd Store (0), add (1) or subtract (-1) the result to idout.

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 will be flagged a sparse and will remain so until you set iadd = 0 to start a new accumulation in idout.

```
call FASTFXQ ( id, *f, n )
```

Interpolate the contents of id to the list of $x$ and $\mu^2$ values that was previously passed to QCDNUM by the call to fastini.

**id** Identifier of an input scratch 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. A dense input buffer is allowed, but wasteful since it contains a lot of information that is not used by fastfxq.

The fast convolution engine is designed for structure function or cross-section calculations but can also be used for simple tasks like efficient pdf interpolation, as is illustrated in the following code (but see also Section 5.8 on fast pdf interpolation).

```
dimension xx(150), qq(150), pdf(150)

call fastini(xx,qq,150,ichk)
call fastepm(iset,0,-1) !copy gluon to buffer #1 (sparse!)
call fastfxq(1,pdf,150) !interpolate gluon
```

### 6.6 Custom Evolution

In Section 5.3 we have described the fillwt routine to generate weight tables (and pdf tables) for the evolution of un-polarised pdfs ($itype = 1$), polarised pdfs ($2$), and fragmentation functions ($3$). But QCDNUM can handle yet another type of evolution, with user-defined evolution kernels (custom evolution, $itype = 4$). For this you have to provide a subroutine (which we call myweight), described below, that generates the weight tables. This routine is passed to QCDNUM as an argument of the custom weight filling routine fillwc, after which the custom evolution becomes available by switching to $itype = 4$.

```
external myweight, func
...
call fillwc( myweight, idmin, idmax, nwords )
...
call evolfg( 4, func, def, iq0, epsi )
...
In Figure 7, we show the listing of a custom weight routine that creates the weight tables of, in fact, un-polarised splitting functions in LO, see Appendix A. The arguments of such a subroutine are as follows.

```fortran
subroutine myweight( w, nw, nwords, idpij, mxord, idum )
imPLICIT double precision (a-h,o-z)
dimension w(*), idPij(7,3)
```

- **w** The QCDNUM store (passed by reference);
- **nw** The number of words available in the store (input, passed by QCDNUM);
- **nwords** Number of words used by the tables (output);
- **idpij** List of weight table identifiers (output);
- **mxord** Maximum perturbative order supported by the tables (output);
- **idum** Not used at present.

The body of the code in Figure 7 shows the steps to be taken in a custom weight routine.

1. Set the maximum order of the custom evolution, here `mxord = 1`;
2. Partition the store into tables by a call to `booktab`. Here are booked two type-1 and two type-2 tables;
3. Branch-out if there is not enough space in the store, as is signalled by a negative value of `nwords` returned by `booktab`
4. Return in `idPij(id,iord)` the identifiers of the various splitting function tables. The first index runs as follows
   
   \[ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \]

   \( P_{qq} \quad P_{gs} \quad P_{sq} \quad P_{gg} \quad P_{+} \quad P_{-} \quad P_{v} \)

   There are only LO splitting functions in the example, with the non-singlet splitting functions all being equal to \( P_{qq} \) (table identifier 101);
5. Generate the weight tables by calls to `makewta`, `makewtb`, `makewrs`, and `makewd`, as is described in Section 6.2. The calls in Figure 7 actually accommodate the splitting functions given in (A.3).

You can have only one set of custom weight tables in memory; a second call to `fillwc` will result in a fatal error message.

A custom evolution must obey the DGLAP evolution equations (2.9) and (2.12) which implies that the evolution must properly split into singlet gluon and non-singlet parts. We remind the reader that the evolution kernels in QCDNUM are defined by convolution with parton number densities, and not parton momentum densities. These kernels can depend on \( x, n_f \) and \( \mu^2 \) and can be stored in tables of types 1, 2, 3 or 4. Note, however, that the \( \mu^2 \) dependence via \( \alpha_s \) is taken care of in the evolution routine `evolsg`, through multiplication of the N^LO tables by \( (\alpha_s/2\pi)^{\ell+1} \).

---

29 The translation from number kernels to momentum kernels is done internally in QCDNUM.
subroutine myweight(w,nw,nwords,idpij,mxord,idum)

C-- w (in) qcdnum store passed by reference
C-- nw (in) number of words available
C-- nwords (out) number of words used < 0 not enough space
C-- idpij (out) list of Pij table identifiers
C-- mxord (out) maximum perturbative order LO,NLO,NNLO
C-- idum not used at present

implicit double precision (a-h,o-z)
dimension w(*), idpij(7,3), itypes(4)

external AChi
external PQQR, PQQS, PQQD ! PQQ
external PQQA, PQQA ! PQG, PGQ
external PGGA, PGGR, PGGS, PGGD ! PGG

call setUmsg('myweight') ! s/r name for error messages

C-1 Max perturbative order
mxord = 1

C-2 Partition
itypes(1) = 2
itypes(2) = 2
itypes(3) = 0
itypes(4) = 0

call BookTab(w,nw,itypes,nwords)

C-3 Not enough space
if(nwords.le.0) return

C-4 Assign table indices
idPij(1,1) = 101 ! PQQ
idPij(2,1) = 201 ! PQG
idPij(3,1) = 102 ! PGQ
idPij(4,1) = 202 ! PGG
idPij(5,1) = 101 ! PPL
idPij(6,1) = 101 ! PMI
idPij(7,1) = 101 ! PVA

C-5 Fill tables

call MakeWRS(w, idPij(1,1), PQQR, PQQS, AChi, 0)
call MakeWtD(w, idPij(1,1), PQQD, AChi)
call MakeWtA(w, idPij(2,1), PQQA, AChi)
call MakeWtA(w, idPij(3,1), PGQA, AChi)
call MakeWtA(w, idPij(4,1), PGGA, AChi)
call MakeWRS(w, idPij(4,1), PGGR, PGGS, AChi, 0)
call MakeWtD(w, idPij(4,1), PGGD, AChi)

C-- Done!
call clrUmsg ! clear s/r name for error messages
return
end

Figure 7: Subroutine that generates weight tables of user-given evolution kernels (LO only, in this example). The subroutine is passed to qcdnum via the routine fillwc, as is described in the text.
6.7 Error Messages in Add-On Packages

When you write an add-on package, the problem arises that QCNUM error messages will be labelled with the name of the QCNUM routine and not with that of the package routine. This can of course become confusing for the user who should be aware only of the package routines, and not what is inside.

One solution is that the package catches errors before QCNUM does, but this would duplicate a good checking mechanism which is already in place. An easier solution is to pass a string to QCNUM 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

\[
\text{call setUmsg('ZMFILLW')}
\]

so that, upon error, the user gets additional information:

```
Error in BOOKTAB ( W, NW, ITYPES, NT, NWDS ) ---> STOP
No x-grid available
Please call GXMAKE
BOOKTAB was called by ZMFILLW
```

The last call in zmfillw is

\[
\text{call clrUmsg}
\]

that wipes the additional message. This is important because downstream QCNUM errors would otherwise appear to have always come from zmfillw.

A call to setUmsg can also be used to control the 1passc error message, see Section 5.4.

7 Acknowledgements

I am of course indebted to the original authors of QCNUM, in particular to M. Virchaux who introduced me to the program in 1991.\(^{30}\) I thank M. Cooper-Sarkar for using preliminary versions of QCNUM17 in her QCD fits and providing important feedback during the development phase of the present version. 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. 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).

\(^{30}\)It was sad to hear that Marc Virchaux passed away in November 2004.
A  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

$$I(x) = \int_x^1 dz \, P(z) \, f\left(\frac{x}{z}\right). \quad (A.1)$$

The LO splitting matrix $P_{ij}^{(0)}$ in (2.14) is written as, in the notation of (2.7),

$$\begin{pmatrix} P_{qq}^{(0)} & P_{qg}^{(0)} \\ P_{gq}^{(0)} & P_{gg}^{(0)} \end{pmatrix} = \begin{pmatrix} P_{qq}^{(0)} & 2n_f P_{qg}^{(0)} \\ P_{gq}^{(0)} & P_{gg}^{(0)} \end{pmatrix}. \quad (A.2)$$

The LO un-polarised splitting functions are given by

$$P_{qq}^{(0)}(x) = \frac{4}{3} \left[ \frac{1 + x^2}{(1-x)^+} + \frac{3}{2} \delta(1-x) \right]$$

$$P_{qg}^{(0)}(x) = \frac{1}{2} \left[ x^2 + (1-x)^2 \right]$$

$$P_{gq}^{(0)}(x) = \frac{4}{3} \left[ \frac{1 + (1-x)^2}{x} \right]$$

$$P_{gg}^{(0)}(x) = 6 \left[ \frac{x}{(1-x)^+} + \frac{1-x}{x} + x(1-x) + \left( \frac{11}{12} - \frac{n_f}{18} \right) \delta(1-x) \right]. \quad (A.3)$$

For the time-like evolution of fragmentation functions, the splitting functions $P_{qg}^{(0)}$ and $P_{gq}^{(0)}$ are exchanged in (A.2) [15]. The ‘+’ prescription in (A.3) is defined by

$$[f(x)]_+ = f(x) - \delta(1-x) \int_0^1 f(z) \, dz \quad (A.4)$$

so that

$$\int_x^1 f(z) [g(z)]_+ \, dz = \int_x^1 [f(z) - f(1)] \, g(z) \, dz - f(1) \int_0^x g(z) \, dz. \quad (A.5)$$

For reference we give the expressions for $I_{qq}$ and $I_{gg}$ obtained from (A.3) and (A.4)

$$I_{qq}^{(0)}(x) = \frac{4}{3} \int_x^1 dz \, \frac{1}{1-z} \left[ (1+z^2)f\left(\frac{x}{z}\right) - 2f(x) \right] + \frac{4}{3} f(x) \left[ \frac{3}{2} + 2 \ln(1-x) \right]$$

$$I_{gg}^{(0)}(x) = 6 \int_x^1 dz \, \frac{1}{1-z} \left[ zf\left(\frac{x}{z}\right) - f(x) \right] + 6 \int_x^1 dz \left[ \frac{1-z}{z} + z(1-z) \right] f\left(\frac{x}{z}\right) + 6 f(x) \left[ \ln(1-x) + \frac{11}{12} - \frac{n_f}{18} \right]. \quad (A.6)$$

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

$$P(x) = A(x) + [B(x)]_+ + R(x)[S(x)]_+ + K(x) \delta(1-x) \quad (A.7)$$
where, of course, not all terms have to be present. The following functions are defined in the logarithmic scaling variable \( y = -\ln(x) \):

\[
h(y) = f(e^{-y}), \quad Q(y) = e^{-y}P(e^{-y}), \quad \bar{A}(y) = e^{-y}A(e^{-y})
\]  
(A.8)

with similar definitions for \( \bar{B} \) and \( \tilde{S} \); however, \( \bar{R}(y) = R(e^{-y}) \) and \( \tilde{K}(y) = K(e^{-y}) \) without a factor \( e^{-y} \) in front. With these definitions (A.1) can be written as

\[
I(y) = \int_0^y du \, Q(u) \, h(y - u) = I_1(y) + I_2(y) + I_3(y) + I_4(y) \quad \text{with}
\]

\[
I_1(y) = \int_0^y du \, \bar{A}(u) \, h(y - u);
\]

\[
I_2(y) = \int_0^y du \, \bar{B}(u) \, [h(y - u) - h(y)] - h(y) \int_0^x dz \, B(z);
\]

\[
I_3(y) = \int_0^y du \, \tilde{S}(u) \, [\tilde{R}(u)h(y - u) - \tilde{R}(0)h(y)] - \tilde{R}(0)h(y) \int_0^x dz \, S(z);
\]

\[
I_4(y) = \tilde{K}(y)h(y),
\]

(A.9)

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 \( u = y - u \) in (A.9).

B Triangular Systems in the DGLAP Evolution

For the non-singlet evolution we have to solve the equation (see Section 3.3)

\[
V a = b.
\]  
(B.1)

The matrix \( V \) 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 \( 4 \times 4 \) example (3.15). This matrix is uniquely determined by storing the first column in a one-dimensional vector \( v \) so that \( V_{ij} = v_{i-j+1} \) for \( i \geq j \), and zero otherwise. Eq. (B.1) is, like any other lower triangular system, iteratively solved by forward substitution

\[
a_1 = b_1 / v_1
\]

\[
a_i = \frac{1}{v_1} \left[ b_i - \sum_{j=1}^{i-1} v_{i-j+1} a_j \right] \quad \text{for} \quad i \geq 2.
\]  
(B.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

\[
\begin{pmatrix}
V_{qq} & V_{qg} \\
V_{gq} & V_{gg}
\end{pmatrix}
\begin{pmatrix}
f \\
g
\end{pmatrix} =
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\begin{pmatrix}
f \\
g
\end{pmatrix} =
\begin{pmatrix}
r \\
s
\end{pmatrix},
\]  
(B.3)
where \( a \) is a short-hand notation for \( V_{qq} \), 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 \( 2 \times 2 \) matrix equation
\[
\begin{pmatrix}
  a_1 & b_1 \\
  c_1 & d_1 \\
\end{pmatrix}
\begin{pmatrix}
  f_1 \\
  g_1 \\
\end{pmatrix}
= \begin{pmatrix}
  r_1 \\
  s_1 \\
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
  f_1 \\
  g_1 \\
\end{pmatrix}
= \frac{1}{a_1 d_1 - b_1 c_1}
\begin{pmatrix}
  d_1 & -b_1 \\
  -c_1 & a_1 \\
\end{pmatrix}
\begin{pmatrix}
  r_1 \\
  s_1 \\
\end{pmatrix}.
\] (B.4)

For \( i \geq 2 \) we have to accumulate the sums
\[
R_i = r_i - \sum_{j=1}^{i-1} [a_{(i+1-j)} f_j + b_{(i+1-j)} g_j]
\]
\[
S_i = s_i - \sum_{j=1}^{i-1} [c_{(i+1-j)} f_j + d_{(i+1-j)} g_j]
\] (B.5)

and solve, for each \( i \), the equations
\[
\begin{pmatrix}
  a_1 & b_1 \\
  c_1 & d_1 \\
\end{pmatrix}
\begin{pmatrix}
  f_i \\
  g_i \\
\end{pmatrix}
= \begin{pmatrix}
  R_i \\
  S_i \\
\end{pmatrix}
\] (B.6)

The operation count of this algorithm is four times that of (B.2), plus some little overhead to solve the \( 2 \times 2 \) matrix equations for each \( i \).

C Zero Mass Structure Functions

C.1 General Formalism

The zero-mass structure functions \( F_2(x, Q^2) \), \( F_L(x, Q^2) \) and \( x F_3(x, Q^2) \) in un-polarised deep inelastic scattering are calculated from (6.1) 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 \( \mu^2 \) and write the singlet/gluon contribution to \( F_2 \) and \( F_L \) as (there is no contribution to \( x F_3 \) since this structure function is a pure non-singlet)
\[
\frac{1}{x} F_i^{(s)}(x, Q^2) = [C_{i,s} \otimes q_s](x, \mu^2) + [C_{i,g} \otimes g](x, \mu^2) \quad i = 2, L.
\] (C.1)

Likewise, non-singlet contributions to the structure functions are given by
\[
\frac{1}{x} F_i^{(ns)}(x, Q^2) = [C_{i,ns} \otimes q_{ns}](x, \mu^2) \quad i = 2, L, 3
\] (C.2)

where the label ‘ns’ stands for the non-singlet indices ‘+’, ‘−’ and ‘ν’ as defined by (2.11). To be precise on notation: \( F_2 = F_2 \), \( F_L = F_L \) and \( F_3 = x F_3 \) in (C.1) and (C.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 [29] 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 $\alpha_s$,

$$C_{i,j}^{\mathrm{NLO}} = \sum_{k=0}^{\ell} a_s^k C_{i,j}^{(k)} \quad i = 2, L, 3 \quad j = g, s, +, -, v$$  \hfill (C.3)

where $\ell = (0, 1, 2)$ denotes (LO, NLO, NNLO) and $a_s = \alpha_s/2\pi$. The LO coefficient functions are either zero or trivial delta functions:

$$C_{2,s}^{(0)} = 0 \quad C_{2,s}^{(0)} = \delta(1 - x) \quad C_{2,ns}^{(0)} = \delta(1 - x)$$

$$C_{L,g}^{(0)} = 0 \quad C_{L,s}^{(0)} = 0 \quad C_{L,ns}^{(0)} = 0$$

$$C_{3,g}^{(0)} = 0 \quad C_{3,s}^{(0)} = 0 \quad C_{3,ns}^{(0)} = \delta(1 - x).$$  \hfill (C.4)

The NLO coefficient functions can be found in [9]. For those at NNLO we refer to [30, 31, 32, 33] and the parametrisations given in [34] and [35].

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

$$C_{L,j}^{\mathrm{NLO}} = \sum_{k=1}^{\ell+1} a_s^k C_{L,j}^{(k)}.$$  \hfill (C.5)

In this way, $C_{L,j}^{(1)}$ is used already at LO (giving a non-zero $F'_L$) and $C_{L,j}^{(2)}$ at NLO. At NNLO the 3-loop coefficient function $C_{L,j}^{(3)}$ is taken from [36]. As stated in [36], this 3-loop calculation applies only to electromagnetic current exchange so that $Z^0$ or $W^\pm$ contributions to $F'_L$ at NNLO are, at present, not available.

### C.2 Renormalisation and Factorisation Scale Dependence

To calculate the renormalisation scale dependence ($\mu_R^2 \neq \mu_F^2$) 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 (C.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 (C.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 ($Q^2 \neq \mu_F^2$), the coefficient functions in (C.3) and (C.5) are replaced by [34, 35]

$$C_{i,j}^{(0)} \rightarrow C_{i,j}^{(0)} \quad \text{and} \quad C_{i,j}^{(k)} \rightarrow C_{i,j}^{(k)} + \sum_{m=1}^k C_{i,j}^{(k,m)} L_F^m \quad k \geq 1,$$  \hfill (C.6)

where $L_F = \ln(Q^2/\mu_F^2)$ and $\mu_F^2 = \mu_R^2$. 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 $C_i = C_{i,ns}$ and a $1 \times 1$ matrix $P = P_{ns}$. In the singlet/gluon sector we have a 2-dimensional row-vector and a $2 \times 2$ matrix that are given by

$$C_i = (C_{i,s} \quad C_{i,g}) \quad \text{and} \quad P = \begin{pmatrix} P_{qq} & P_{qg} \\ P_{gq} & P_{gg} \end{pmatrix}.$$
In this vector notation, the functions $C_{i,j}^{(k,m)}$ in (C.6) are written as

\begin{align*}
C_{i,j}^{(1,1)} &= C_{i,j}^{(0)} \otimes P^{(0)} \\
C_{i,j}^{(2,1)} &= C_{i,j}^{(0)} \otimes P^{(1)} + C_{i,j}^{(1)} \otimes [P^{(0)} - \beta_0 I] \\
C_{i,j}^{(2,2)} &= \frac{1}{2} C_{i,j}^{(1,1)} \otimes [P^{(0)} - \beta_0 I] \\
C_{i,j}^{(3,1)} &= C_{i,j}^{(0)} \otimes P^{(2)} + C_{i,j}^{(1)} \otimes [P^{(1)} - \beta_1 I] + C_{i,j}^{(2)} \otimes [P^{(0)} - 2\beta_0 I] \\
C_{i,j}^{(3,2)} &= \frac{1}{2} \left[ C_{i,j}^{(1,1)} \otimes [P^{(1)} - \beta_1 I] + C_{i,j}^{(2,1)} \otimes [P^{(0)} - 2\beta_0 I] \right] \\
C_{i,j}^{(3,3)} &= \frac{1}{3} C_{i,j}^{(2,2)} \otimes [P^{(0)} - 2\beta_0 I].
\end{align*}

(C.7)

For $F_2$, $F_L$ and $xF_3$, the coefficients are calculated up to $C_{i,j}^{(2,2)}$. For $F'_L$, on the other hand, all coefficients in (C.7) are computed. Note, however, that quite some convolutions are trivial because the LO coefficient functions are either zero or $\delta$-functions, see (C.4).

As mentioned above, the expression (C.6) applies only when $\mu_F^2 = \mu_R^2$. It is therefore not possible to vary both scales $\mu_F^2$ and $Q^2$ at the same time.

### C.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 convolution engine described in Section 6.

The list of subroutines is given in Table 6. Note that error messages are, in most cases, issued by the underlying QCDNUM routines and not by the ZMSTF routine itself.

<table>
<thead>
<tr>
<th>Subroutine or function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZMWORDS (*ntotal, *nused)</td>
<td>Words available, used</td>
</tr>
<tr>
<td>ZMFILLW (*nused)</td>
<td>Fill weight tables</td>
</tr>
<tr>
<td>ZMDUMPW (lun, 'filename')</td>
<td>Dump weight tables</td>
</tr>
<tr>
<td>ZMREADW (lun, 'filename', *nused, *ierr)</td>
<td>Read weight tables</td>
</tr>
<tr>
<td>ZMDEFQ2 (a, b)</td>
<td>Define $Q^2$</td>
</tr>
<tr>
<td>ZMABVAL (*a, *b)</td>
<td>Retrieve a and b coefficients</td>
</tr>
<tr>
<td>ZMQFRMU (gmu2)</td>
<td>Convert $\mu_F^2$ to $Q^2$</td>
</tr>
<tr>
<td>ZMUFRMQ (Q2)</td>
<td>Convert $Q^2$ to $\mu_F^2$</td>
</tr>
<tr>
<td>ZSWITCH (iset)</td>
<td>Switch pdf set</td>
</tr>
<tr>
<td>ZMSTFUN (istf, def, x, Q2, *f, n, ichk)</td>
<td>Structure functions</td>
</tr>
</tbody>
</table>

Output arguments are pre-fixed with an asterisk (*).
However, the calling ZMSTF routine is mentioned in the error message so that you know where it came from.

```fortran
call ZMWORDS ( *ntotal, *nused )
```

**ntotal**  Number of words available in the ZMSTF store (nzmstor in zmstf.inc).

**nused**  Number of words used (set to 0 before the call to zmfillw or zmreadw).

```fortran
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 store is returned in **nused**. If you get an error message that the internal store 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 \( x-\mu^2 \) 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).

```fortran
call ZMDUMPW ( lun, 'filename' )
```

Dump the weights in memory via logical unit number **lun** to a disk file. The dump is unformatted so that the weight file cannot be exchanged across machines.

```fortran
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 store (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 \( x-\mu^2 \) 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.

```fortran
call ZMDEFQ2 ( a, b )
```

Define the relation between the factorisation scale \( \mu_F^2 \) 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 $0.1 \leq a \leq 10$ and $-100 \leq b \leq 100$.

A call to zmabval(a,b) reads the coefficients back from memory. To convert between the scales use:

$$Q^2 = zmqfrmu(qmu2)$$
$$qmu2 = zmufrmq(Q^2)$$

```
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 (5–9). 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.

<table>
<thead>
<tr>
<th>Contribution</th>
<th>LO</th>
<th>NLO</th>
<th>NNLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quark and gluon</td>
<td>±101</td>
<td>±102</td>
<td>±103</td>
</tr>
<tr>
<td>Quark only</td>
<td>±201</td>
<td>±202</td>
<td>±203</td>
</tr>
<tr>
<td>Gluon only</td>
<td>±301</td>
<td>±302</td>
<td>±303</td>
</tr>
</tbody>
</table>

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).
D Heavy Quark Structure Functions

D.1 General Formalism

A NLO calculation of the heavy quark contributions to the $F_2$ and $F_L$ structure functions in deep inelastic charged lepton-proton scattering is given in [16]. 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 $n_f = (3, 4, 5)$ for $h = (c, b, t)$ [37].

A heavy flavour contribution to $F_2$ or $F_L$ is calculated from

$$F_h^k(x, Q^2) = \frac{\alpha_s}{2\pi} \left\{ e_h^2 g \otimes C_{k,g}^{(0)} + \frac{\alpha_s}{2\pi} \left( e_h^2 g \otimes C_{k,g}^{(1)} + e_h^2 q_h \otimes C_{k,q}^{(1)} + q_p \otimes D_{k,q}^{(1)} \right) \right\} ,$$

where $e_h$ is the charge of the heavy quark (in units of the positron charge), $g$ is the gluon density, $q_s$ is the singlet density and

$$q_p = \sum_{i=1}^{n_f} e_i^2 (q_i + \bar{q}_i)$$

is the charge-weighted proton quark distribution for $n_f$ light flavours. The first term in (D.1) is the LO contribution from the photon-gluon fusion process $\gamma^* g \rightarrow h\bar{h}$. The last three terms correspond to the NLO sub-process $\gamma^* g \rightarrow h\bar{h}g$ and $\gamma^* q \rightarrow h\bar{h}q$.\(^{31}\) For the heavy quark coefficient functions $C$ and $D$ in (D.1) we refer to [16].\(^{32}\)

In terms of a number density $f(x, \mu^2)$, the convolution integrals in (D.1) are defined by

$$f \otimes C = \int_{ax}^{1} \frac{dz}{z} z f(z, \mu^2) C(x/z, Q^2, \mu^2, m_h^2)$$

where $a = 1 + 4m_h^2/Q^2$ and $\mu^2$ 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 $\gamma^*p$ 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 (D.2). It turns out that the dependence of the coefficient functions on the relation between $Q^2$ and $\mu^2$ 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 $a_s = \alpha_s/2\pi$ in (D.1) are replaced by the Fourier expansion (2.17), truncated to $a_s$ in LO, and to $a_s^2$ in NLO. Note that you can vary either $\mu_R^2$ or $Q^2$ with respect to $\mu_F^2$, but not both at the same time.

\(^{31}\)In the LO and the first two NLO terms the virtual photon couples to the heavy quark, hence the factor $e_h^2$ in (D.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, $q_p$, of light quark distributions.

\(^{32}\)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 analytical form. Note that in [16] the coefficient functions are convoluted with parton momentum densities and not with number densities.
The convolution integral (D.2) is not of the general form (6.1): (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 $\chi/z$. 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.

D.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 7. We will only describe here the routines `hqfillw` and `hqstfun`, the other ones being similar to those in the ZMSTF package.

<table>
<thead>
<tr>
<th>Subroutine or function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HQWORDS ( *ntotal, *nused )</td>
<td>Words available, used</td>
</tr>
<tr>
<td>HQFILLW ( istf, qmass, aq, bq, *nused )</td>
<td>Fill weight tables</td>
</tr>
<tr>
<td>HQDUMPW ( lun, 'filename' )</td>
<td>Dump weight tables</td>
</tr>
<tr>
<td>HQREADW ( lun, 'filename', *nused, * ierr )</td>
<td>Read weight tables</td>
</tr>
<tr>
<td>HQPARMS ( *qmass, *aq, *bq )</td>
<td>Retrieve parameters</td>
</tr>
<tr>
<td>HQQFRMU ( qmu2 )</td>
<td>Convert $\mu^2$ to $Q^2$</td>
</tr>
<tr>
<td>HQMUFQRQ ( Q2 )</td>
<td>Convert $Q^2$ to $\mu^2_F$</td>
</tr>
<tr>
<td>HSNSWITCH ( iset )</td>
<td>Switch pdf set</td>
</tr>
<tr>
<td>HQSTFUN ( istf, icbt, def, x, Q2, *f, n, ichk )</td>
<td>Structure functions</td>
</tr>
</tbody>
</table>

Output arguments are pre-fixed with an asterisk (*).

Call the HQFILLW subroutine before anything else.

```latex
\text{call HQFILLW ( istf, qmass, aq, bq, *nused )}
```

Select structure function: $1 = F_L, 2 = F_2$ and $3 = \text{both}$.

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.

Defines the relation $Q^2 = a\mu_F^2 + b$.

Gives, on exit, the number of words used in the store.

You will get a fatal error if the store 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)`.

73
call HQSTFUN ( istf, icbt, def, x, Q2, *f, n, ichk )

Calculate the heavy quark contribution to a structure function.

istf   Calculate $F_1$ (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 (6.8).
x, Q2   Input arrays containing a list of $x$ and $Q^2$ (not $\mu^2$) values.
f       Output array containing the list of structure functions.
n       Number of items in x, Q2 and f.
icchk   If set to zero, hqstfun will return a null value when $x$ or $\mu^2$ are outside the grid boundaries; 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 prepand 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).

```fortran
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 zmstf(2, proton, x, Q2, F2dus, 100, ichk)
call hqstf(2, 1, proton, x, Q2, F2c , 100, ichk)
do i = 1,100
   F2p(i) = F2dus(i) + F2c(i)
endo
```

33For technical reasons a cut $Q^2 > 0.5$ GeV$^2$ is also imposed.
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. Here is an up-to-date list of all releases and updates.

<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>17-00/06</td>
<td>10-07-12</td>
<td>In the MFNS, the function asfunc evolved $\alpha_s$ in the FFNS. The QCDNUM internal $\alpha_s$ tables were not affected by this bug.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>17-00/05</td>
<td>10-04-12</td>
<td>Access to version number and qcdnum.inc parameters (via getint).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New routine mixfns to set the mixed flavour number scheme.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New routine set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New function lpassc to check if a point passes the cuts.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New routines pdf1st and pdftab for fast pdf interpolations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove the mpt0 limit on the number of interpolations in QCDNUM, ZMSTF and HQSTF. In fastini the mpt0 limit still exists.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increase storage sizes nwf0, nzmstor and nhqstor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZMSTF: New routine zmwords gives access to storage size/use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZMSTF: Possibility to separately calculate gluon or quark contributions to structure functions, order by order (with zmslowf).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HQSTF: New routine hqwords gives access to storage size/use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HQSTF: When using external pdfs (iset = 5–9), the availability of unpolarised pdfs (iset = 1) was imposed. Bug now fixed.</td>
</tr>
<tr>
<td>17-00/04</td>
<td>18-07-11</td>
<td>Increase storage sizes nwf0, nzmstor and nhqstor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set $Q^2 = 0.25$ GeV$^2$ in the HQSTF coefficient functions when the input value of $Q^2 &lt; 0$. This avoids problems in hqfillw if the low end of the $\mu^2$ grid maps onto negative $Q^2$.</td>
</tr>
<tr>
<td>17-00/03</td>
<td>30-03-11</td>
<td>Rename splitting/coefficient functions in QCDNUM and ZMSTF to avoid name clashes with QCDNUM16 and—more important—with LHAPDF (which has QCDNUM16 inside).</td>
</tr>
<tr>
<td>17-00/02</td>
<td>07-10-10</td>
<td>Comments from the CPC referees included in the write-up.</td>
</tr>
<tr>
<td>17-00/01</td>
<td>03-09-10</td>
<td>Adjust internal cuts on the $\alpha_s$ evolution to allow for evolution to lower $Q^2$. The cuts in the original release were set too tight.</td>
</tr>
<tr>
<td>17-00/00</td>
<td>08-05-10</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

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


[37] E. Laenen, private communication.
Index

B-splines, 16–17
backward evolution, 22–23, 31
beta-functions, 6
Bjorken-x variable, 7

convolution integrals in QCDNUM, 18, 44–45
convolution weights, 18
custom evolution, 61–62

DGLAP evolution equations, 7
discontinuities in αs evolution, 7, 12–13, 27
discontinuities in pdf evolution, 13–14
e± basis pdfs, 10
evolution of αs, 6–7
evolution of heavy quark pdfs, 14–15

factorisation scale µ2, 8
factorisation scale dependence, 68
F2L structure function, 68
FFNS fixed flavour number scheme, 12, 37
flavour number schemes, 12, 15, 37
flavour thresholds
on the factorisation scale, 9
on the renormalisation scale, 7, 9, 13
forward substitution, 66–67
Fourier convolution, 18
fragmentation functions, see time-like evolution
Gauss quadrature in QCDNUM, 46
accuracy parameter of, 31
generalised mass (GM) schemes, 45
HQSTF package, 73–74
indexing of e± basis, 43
indexing of q, ¯q basis, 40
interpolation mesh, 22
interpolation of pdfs, 42–43, 61
ket notation, 10
MBUTIL package, 24
Mellin convolution, 8
mesh point, see interpolation mesh

MFNS mixed flavour number scheme, 15, 37
MS scheme, 5
multiple convolution, 18, 19, 50, 59, 60
multiple evolution grid, 22, 27–29, 32
non-singlet evolution, 9
non-singlet quark density, 8, 10
null value, 31
number of active flavours nf, 6
number at threshold, 9, 27, 55, 60
parton density function, pdf, 6
parton luminosity, 19
pdf type, pdf set, 35, 42, 61
PDG convention, 26
PEGASUS program, 15, 27
polarised evolution, 9

QCDNUM program
example job, 24–27
execution speed, 29
history, 5
list of subroutines, 30, 47, 52, 56
parameters in qcdnum.inc, 24
program hang-up, 29
web site, 24

QCDNUM convolution engine
booktab, 47
clrumsg, 64
copywgt, 50
efromqq, 54
fcrossf, 53
fcrossk, 53
getalfn, 54
getwpar, 51
idspfun, 49
makewrs, 49
makewta, 48
makewtb, 49
makewtd, 49
makewtx, 49
nflavor, 54
qqfrome, 54
scalewt, 49

79
renormalisation scale $\mu_R^2$, 6
renormalisation scale dependence of pdfs, 9–10
of structure functions, 68, 72
rescaling variable $\chi$, 45
scale parameter $\Lambda$, 7
singlet quark density, 8
singlet-gluon evolution, 8
singlet/non-singlet basis $e^\pm$, 10
spline interpolation, 15–17, 43
spline oscillation, 16, 22–23, 31, 40, 43
splitting functions
at leading order, 65
perturbative expansion of, 9
singularities in, 46, 65–66
symmetries in, 8
structure functions
heavy flavour contributions, 72–73
zero-mass structure functions, 67–69
sum rule integrals, 26
time-like evolution, 9, 65
Toeplitz matrix, 19
truncation prescription, 10, 54, 68
\(t\)-variable, 18
types of weight table, 46

un-polarised evolution, 7

VFNS variable flavour number scheme, 12, 37

weights, see convolution weights
Wilson coefficient, 67

\(y\)-variable, 18

ZMSTF package, 69–71