A Higher Order Perturbative Parton Evolution Toolkit (HOPPET) version 1.1.5

G. P. Salam and J. Rojo
LPTHE,
UPMC – Univ. Paris 6,
Université Paris Diderot – Paris 7,
CNRS UMR 7589,
75252 Paris cedex 05, France

Abstract

This document describes HOPPET, a Fortran 95 package for carrying out DGLAP evolution and other common manipulations of parton distribution functions (PDFs). The PDFs are represented on a grid in x–space so as to avoid limitations on the functional form of input distributions. Good speed and accuracy are obtained through the representation of splitting functions in terms of their convolution with a set of piecewise polynomial basis functions, and Runge-Kutta techniques are used for the evolution in Q. Unpolarised evolution is provided to NNLO, including heavy-quark thresholds in the $\overline{\rm MS}$ scheme, and longitudinally polarised evolution to NLO. The code is structured so as to provide simple access to the objects representing splitting functions and PDFs, making it possible for a user to extend the facilities already provided. A streamlined interface is also available, facilitating use of the evolution part of the code from F77 and C/C++.

Program Summary

Title of program: HOPPET

Version: 1.1.5

Catalogue identifier:

Program obtainable from: `http://projects.hepforge.org/hoppet/`

Distribution format: compressed tar file

E-mail: `salam@lpthe.jussieu.fr`, `rojo@lpthe.jussieu.fr`

License: GNU Public License

Computers: all

Operating systems: all

Program language: Fortran 95

Memory required to execute: $\lesssim$ 10 MB

Other programs called: none

External files needed: none

Number of bytes in distributed program, including test data etc.: $\sim$ 350 kB

Keywords: unpolarised and longitudinally polarised parton space-like distribution functions (PDFs), DGLAP evolution equations, x–space solutions.

Nature of the physical problem: Solution of the DGLAP evolution equations up to NNLO (NLO) for unpolarised (longitudinally polarised) PDFs, and provision of tools to facilitate manipulation (convolutions, etc.) of PDFs with user-defined coefficient and splitting functions.

Method of solution: representation of PDFs on a grid in x, adaptive integration of splitting functions to reduce them to a discretised form, obtaining fast convolutions that are equivalent to integration with an interpolated form of the PDFs; Runge Kutta solution of the Q evolution, and its caching so as to speed up repeated evolution with different initial conditions.

Restrictions on complexity of the problem: PDFs should be smooth on the scale of the discretisation in x.

Typical running time: a few seconds for initialisation, then $\sim$ 10 ms for creating a tabulation with a relative accuracy of 10^{-4} from a new initial condition (on a 3.4 GHz Pentium IV processor). Further details in section 9.2.

Contents

1 Introduction

There has been considerable discussion over the past years (e.g. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) of numerical solutions of the Dokshitzer-Gribov-Lipatov-Altarelli-Parisi (DGLAP) equation [11] for the Quantum Chromodynamics (QCD) evolution of parton distribution functions (PDFs).

The DGLAP equation [11] is a renormalisation group equation for the quantity $q_i(x, Q^2)$, the density of partons of type (or flavour) i carrying a fraction x of the longitudinal momentum of a hadron, when resolved at a scale Q. It is one of the fundamental equations of perturbative QCD, being central to all theoretical predictions for hadron-hadron and lepton-hadron colliders.

Technically, it is a matrix integro-differential equation,

$$\frac{\partial q_i(x,Q^2)}{\partial \ln Q^2} = \frac{\alpha_s\left(Q^2\right)}{2\pi}\int_x^1 \frac{dz}{z} P_{ij}\left(z,\alpha_s\left(Q^2\right)\right) q_j\left(\frac{x}{z},Q^2\right) , \tag{1}$$

whose kernel elements $P_{ij}(z, Q^2)$ are known as splitting functions, since they describe the splitting of a parton of kind j into a parton of kind i carrying a fraction z of the longitudinal momentum of j. The parton densities themselves $q_i(x, Q^2)$ are essentially non-perturbative, since they depend on physics at hadronic mass scales $\lesssim 1$ GeV, where the QCD coupling is large. On the other hand the splitting functions are given by a perturbative expansion in the QCD coupling $\alpha_s(Q^2)$. Thus given partial experimental information on the parton densities[1] — for example over a limited range of Q, or for only a subset of parton flavours

[1] Of course it is not the parton densities, but rather structure functions, which can be derived from them perturbatively, that are measured experimentally.

— the DGLAP equations can be used to reconstruct the parton densities over the full range of Q and for all flavours.

The pivotal role played by the DGLAP equation has motivated a considerable body of literature discussing its numerical solution [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. There exist two main classes of approaches: those that solve the equation directly in x–space and those that solve it for Mellin transforms of the parton densities, defined as

$$q_N\left(N,Q^2\right) = \int_0^1 dx x^N q_i(x,Q^2)\ , \tag{2}$$

and subsequently invert the transform back to x–space. Recently, a novel approach has been proposed which combines advantages of the $N-$space and $x-$space methods [8]. $N-$space based methods are of interest because the Mellin transform converts the convolution of eq. (1) into a multiplication, resulting in a continuum of independent matrix differential (rather than integro-differential) equations, one for each value of N, making the evolution more efficient numerically.

The drawback of the Mellin method is that one needs to know the Mellin transforms of both the splitting functions and the initial conditions. There can also be subtleties associated with the inverse Mellin transform. The x–space method is in contrast more flexible, since the inputs are only required in x–space; however it is generally considered to less efficient numerically, because of the need to carry out the convolution in eq. (1).

To understand the question of efficiencies one should analyse the number of operations needed to carry out the evolution. Assuming that one needs to establish the results of the evolution at N_x values of x, and N_Q values of Q, one essentially needs $\mathcal{O}\left(N_x^2 N_Q\right)$ operations with an x–space method, where the N_x^2 factor comes from the convolutions. In the Mellin-space method, one needs $\mathcal{O}\left(N_x N_Q M\right)$ operations, where M is the number of points used for Mellin inversion. One source of drawback of the x–space method is that, nearly always, $N_x \sim \ln 1/x_{\rm min}$ and so the method scales as the square of $\ln 1/x_{\rm min}$, where the Mellin method is linear (and M can be kept roughly independent of $x_{\rm min}$).

The other issue relates to how one goes to higher numerical integration and interpolation orders. In x–space methods one tends to choose x values that are uniformly distributed (be it in $\ln 1/x$ or some other more complex function) — this limits one to higher-order extensions of the Trapezium and Simpson-rule type integrations, whose order in general is $n_p - 1$ where n_p is the number of points used for the integration. The precision of the integration is given by $(\delta x)^{n_p}$ where δx is the grid spacing. Higher n_p improves the accuracy, but typically n_p can not be taken too large because of large cancellations between weights that arise for large n_p. In the Mellin method one is free to position the M points as one likes, and one can then use Gaussian type integration [5, 3, 9]; using n_p points one manages to get a numerical order $2n_p - 1$, i.e. accuracy $(\delta x)^{2n_p}$, and furthermore the integration weights do not suffer from cancellations at large n_p, allowing one to increase n_p, and thus the accuracy, quite considerably.

Despite it being more difficult to obtain high accuracy with x–space methods, their greater flexibility means that they are widespread, serving as the basis of the well-known QCDNUM program [1], and used also by the CTEQ [12] and MRST/MSTW [13] global

fitting collaborations. Higher-order methods in x–space have been developed in [2, 4, 6, 7, 14], and more recently have been incorporated also in QCDNUM.

HOPPET, the program presented here, uses higher-order methods both for the x-integrations and Q evolution. It combines this with multiple grids in x–space: a high-density grid at large x where it is hardest to obtain good accuracy, and coarser grids at smaller x where the smoothness of the PDFs facilitates the integrations. One of the other crucial features of the program is that it pre-calculates as much information as possible, so as to reduce the evolution of a new PDF initial condition to a modest set of addition and multiplication operations. Additionally, the program provides access to a range of low and medium-level operations on PDFs which should allow a user to extend the facilities already provided.

The functionality described in this article has been present in HOPPET's predecessors for several years (they were available on request), but had never been documented. Those predecessors have been used in a number of different contexts, like resummation of event shapes in DIS [14], automated resummation of event shapes [15], studies of resummation in the small-x limit [16], and in a posteriori inclusion of PDFs in NLO final-state calculations [17, 18], as well as used for benchmark comparisons with Pegasus [3] in [19]. Since the code had not hitherto been released in a documented form, it is the authors' hope that availability of this documentation may make the package somewhat more useful.

This manual is structured as follows: section 2 briefly summarises the perturbative QCD ingredients contained in HOPPET, while section 3 describes the numerical techniques used to solve the DGLAP equation. sections 4–7 present in detail the capabilities of the HOPPET package with its general F95 interface, with emphasis on those aspects that can be adapted by a user to tailor it to their own needs. section 8 describes a streamlined interface to HOPPET which embodies its essential capabilities in a simple interface available in a variety of programming languages: F77 and C/C++. Finally, section 9 presents a detailed quantitative study of the performance of HOPPET, and in the final section we conclude. A set of appendices contain various example programs, both for the general and the streamlined interfaces, a reference guide with the most important HOPPET modules, details on technical aspects and a set of useful tips on Fortran 95.

A reader whose interest is to use HOPPET to perform fast and efficient evolution of PDFs may wish to skip sections 4 to 7 and move directly to section 8, which describes the user-friendlier streamlined interface, accessible from F95, F77 and C/C++, and which contains the essential functionalities of HOPPET. He/she is also encouraged to go through the various example programs which contain detailed descriptions and explanations. On the other hand, a reader interested in the more flexible and general functionalities of HOPPET should also consult sections 4 to 7.

Note that throughout this documentation, a PDF refers always to a momentum density rather than a parton density, that is, when we refer to a gluon, we mean $xg(x)$ rather than $g(x)$, the same convention as used in the LHAPDF PDF library [20].

2 Perturbative evolution in QCD

First of all we set up the notation and conventions that are used throughout HOPPET. The DGLAP equation for a non-singlet parton distribution reads

$$\frac{\partial q(x,Q^2)}{\partial \ln Q^2} = \frac{\alpha_s\left(Q^2\right)}{2\pi}\int_x^1 \frac{dz}{z} P(z,\alpha_s\left(Q^2\right)) q\left(\frac{x}{z},Q^2\right) \equiv \frac{\alpha_s\left(Q^2\right)}{2\pi} P(x,\alpha_s\left(Q^2\right)) \otimes q\left(x,Q^2\right) \ . \tag{3}$$

The related variable $t \equiv \ln Q^2$ is also used in various places in HOPPET. The splitting functions in eq. (3) are known up to NNLO in the unpolarised case [21, 22, 23]:

$$P(z,\alpha_s\left(Q^2\right)) = P^{(0)}(z) + \frac{\alpha_s\left(Q^2\right)}{2\pi} P^{(1)}(z) + \left(\frac{\alpha_s\left(Q^2\right)}{2\pi}\right)^2 P^{(2)}(z) \ , \tag{4}$$

and up to NLO [24, 25] in the polarised case. The generalisation to the singlet case is straightforward, as it is to the case of time-like evolution[2], relevant for example for fragmentation function analysis, where partial NNLO results are also available [28].

As with the splitting functions, all perturbative quantities in HOPPET are defined to be coefficients of powers of $\alpha_s/2\pi$. The one exception is the β-function coefficients of the running coupling equation:

$$\frac{d\alpha_s}{d\ln Q^2} = \beta\left(\alpha_s\left(Q^2\right)\right) = -\alpha_s(\beta_0\alpha_s + \beta_1\alpha_s^2 + \beta_2\alpha_s^3) \ . \tag{5}$$

The evolution of the strong coupling and the parton distributions can be performed in both the fixed flavour-number scheme (FFNS) and the variable flavour-number scheme (VFNS). In the VFNS case we need the matching conditions between the effective theories with n_f and n_f+1 light flavours for both the strong coupling $\alpha_s\left(Q^2\right)$ and the parton distributions at the heavy quark mass threshold m_h^2.

These matching conditions for the parton distributions receive non-trivial contributions at higher orders. In the $\overline{\rm MS}$ (factorisation) scheme, for example, these begin at NNLO:[3] for light quarks $q_{l,i}$ of flavour i (quarks that are considered massless below the heavy quark mass threshold m_h^2) the matching between their values in the n_f and n_f+1 effective theories reads:

$$q_{l,i}^{(n_f+1)}(x,m_h^2) \ = \ q_{l,i}^{(n_f)}(x,m_h^2) + \left(\frac{\alpha_s(m_h^2)}{2\pi}\right)^2 A_{qq,h}^{\rm ns,(2)}(x) \otimes q_{l,i}^{(n_f)}(x,m_h^2) \ , \tag{6}$$

where $i = 1, \ldots n_f$, while for the gluon distribution and the heavy quark PDF q_h one has

[2] The general structure of the relation between space-like and time-like evolution and splitting functions has been investigated in [21, 26, 27, 28, 29, 30, 31].

[3] In a general scheme they would start at NLO.

a coupled matching condition:

$$
\begin{aligned}
g^{(n_f+1)}(x,m_h^2) &= g^{(n_f)}(x,m_h^2) \\
&\quad + \left(\frac{\alpha_s(m_h^2)}{2\pi}\right)^2 \left[A^{S,(2)}_{\rm gq,h}(x)\otimes\Sigma^{(n_f)}(x,m_h^2) + A^{S,(2)}_{\rm gg,h}(x)\otimes g^{(n_f)}(x,m_h^2)\right] , \\
(q_h+\bar{q}_h)^{(n_f+1)}(x,m_h^2) &= \left(\frac{\alpha_s(m_h^2)}{2\pi}\right)^2 \left[\tilde{A}^{S,(2)}_{\rm hq}(x)\otimes\Sigma^{(n_f)}(x,m_h^2) + \tilde{A}^{S,(2)}_{\rm hg}(x)\otimes g^{(n_f)}(x,m_h^2)\right] ,
\end{aligned}
\tag{7}
$$

with $q_h = \bar{q}_h$, and the singlet PDF $\Sigma(x,Q^2)$ is defined in Table 1. The NNLO matching coefficients were computed in [32][4]. Notice that the above conditions will lead to small discontinuities of the PDFs in its evolution in Q^2, which are cancelled by similar matching terms in the coefficient functions resulting in continuous physical observables. In particular, the heavy quark PDFs start from a non-zero value at threshold at NNLO, which sometimes can even be negative.

The corresponding NNLO relation for the matching of the $\overline{\rm MS}$ coupling constant at the heavy quark threshold m_h^2 is given by

$$
\alpha_s^{(n_f+1)}(m_h^2) \;=\; \alpha_s^{(n_f)}(m_h^2) + C_2\left(\frac{\alpha_s^{(n_f)}(m_h^2)}{2\pi}\right)^3 \;, \tag{8}
$$

where the matching coefficient C_2 was computed in [33]. The value of C_2 and the form of the matching coefficients in eqs. (6,7) depend on the scheme used for the quark masses; by default in HOPPET quark masses are taken to be pole masses, though the option exists for the user to supply and have thresholds crossed at $\overline{\rm MS}$ masses.

Both evolution and threshold matching preserve the momentum sum rule

$$
\int_0^1 dx\; x\left(\Sigma(x,Q^2)+g(x,Q^2)\right) = 1\,, \tag{9}
$$

and valence sum rules

$$
\int_0^1 dx\;\left[q(x,Q^2)-\bar{q}(x,Q^2)\right] = \begin{cases} 1, & \text{for } q=d \text{ (in proton)} \\ 2, & \text{for } q=u \text{ (in proton)} \\ 0, & \text{other flavours} \end{cases} \tag{10}
$$

as long as they hold at the initial scale (occasionally not the case, e.g. in modified LO sets for Monte Carlo generators [34]).

The default basis for the PDFs, called the `human` representation in HOPPET, is such that the entries in an array `pdf(-6:6)` of PDFs correspond to:

$$
\begin{aligned}
\bar{t}=-6\,,\bar{b}=-5\,,\bar{c}=-4\,,\bar{s} &= -3\,,\bar{u}=-2\,,\bar{d}=-1\,, \\
g &= 0\,, \\
d=1\,,u=2\,,s=3\,,c &= 4\,,b=5\,,t=6\,.
\end{aligned}
\tag{11}
$$

[4]The authors are thanked for the code corresponding to the calculation.

i	name	q_i
$-6 \ldots -(n_f+1)$	q_i	q_i
$-n_f \ldots -2$	$q^-_{\mathrm{NS},i}$	$(q_i - \bar{q}_i) - (q_1 - \bar{q}_1)$
-1	q^V_{NS}	$\sum_{j=1}^{n_f}(q_j - \bar{q}_j)$
0	g	gluon
1	Σ	$\sum_{j=1}^{n_f}(q_j + \bar{q}_j)$
$2 \ldots n_f$	$q^+_{\mathrm{NS},i}$	$(q_i + \bar{q}_i) - (q_1 + \bar{q}_1)$
$(n_f+1) \ldots 6$	q_i	q_i

Table 1:
The evolution representation (called `evln` in HOPPET) of PDFs with n_f active quark flavours in terms of the `human` representation.

This representation is the same as that used in the `LHAPDF` library [20]. However, this representation leads to a complicated form of the evolution equations. The splitting matrix can be simplified considerably (made diagonal except for a 2×2 singlet block) by switching to a different flavour representation, which is named the `evln` representation, for the PDF set, as explained in detail in [35, 36]. This representation is described in Table 1.

In the `evln` basis, the gluon evolves coupled to the singlet PDF Σ, and all non-singlet PDFs evolve independently. Notice that the representations of the PDFs are preserved under linear operations, so in particular they are preserved under DGLAP evolution. The conversion from the `human` to the `evln` representations of PDFs requires that the number of active quark flavours n_f be specified by the user, as described in section 5.1.2.

In HOPPET unpolarised DGLAP evolution is available up to NNLO in the $\overline{\mathrm{MS}}$ scheme, while for the DIS scheme only evolution up to NLO is available, but without the NLO heavy-quark threshold matching conditions. For polarised evolution only the $\overline{\mathrm{MS}}$ scheme is available. The variable `factscheme` takes different values for each factorisation scheme:

`factscheme`	Evolution
1	unpolarised $\overline{\mathrm{MS}}$ scheme
2	unpolarised DIS scheme
3	polarised $\overline{\mathrm{MS}}$ scheme

Note that mass thresholds are currently missing in the DIS scheme.

3 Numerical techniques

We briefly introduce now the numerical techniques used to perform parton evolution: the discretisation of PDFs and their convolutions with splitting functions on a grid in x, and the subsequent DGLAP evolution in Q^2.

The first aspect that we discuss is how to represent PDFs and associated convolutions in terms of an interpolating grid in $x-$space. These techniques can be applied to a range

of problems that involve convolutions, so are not restricted to parton distributions. Then in later sections we will describe how to obtain the solution of the DGLAP evolution equations.

3.1 Higher order matrix representation

Given a set of N_x grid points $y_\alpha = \ln 1/x_\alpha$, labelled by an index α and (for later convenience) a uniform grid spacing, $y_\alpha = \alpha\delta y$, one can approximate a parton distribution function $xq\,(x,t)$ by interpolating the PDF at the grid points,[5]

$$xq(y = \ln 1/x, t) = \sum_\alpha w_\alpha(y) q_\alpha(t)\,, \tag{12}$$

where $w_\alpha(y)$ are the interpolation weights, we have defined

$$q_\alpha(t) \equiv x_\alpha q(y_\alpha, t)\ , \tag{13}$$

and the sum over α runs over $n+1$ points in the vicinity of y for an interpolation order n. Note that Greek indices represent the y dimension, while Roman indices are used to represent the flavour dimension.

The convolution of a single-flavour PDF with a splitting function $P(z,t)$ can be written as

$$(P \otimes q)(y,t) = \sum_\alpha \omega_\alpha(y)(P \otimes q)_\alpha(t)\ , \tag{14}$$

where we have replaced the convolution by its grid representation,

$$(P \otimes q)_\alpha(t) = \sum_\beta P_{\alpha\beta}(t)\, q_\beta(t)\,, \tag{15}$$

where β runs over $\mathcal{O}\,(N_x)$ points of the grid and we have defined

$$P_{\alpha\beta}(t) = \int_{e^{-y_\alpha}}^1 dz\, P(z,t)\, w_\beta(y_\alpha + \ln z)\ . \tag{16}$$

A virtue of having a uniform grid in $y = \ln 1/x$ is that the interpolation functions can be arranged to have a structure $w_\alpha(y) = w(y - y_\alpha)$ (where $w(y)$ is non-zero for $0 \le y < n\delta y$), so that $P_{\alpha\beta}$ just depends on $\alpha - \beta$, and can be rewritten $\mathcal{P}_{\alpha-\beta}$. A slight subtlety arises at large x, where if one writes $w_\alpha(y) = w(y - y_\alpha)$ one is effectively assuming an interpolation that uses identically zero interpolation points for $x \ge 1$ [6, 10], even though $xq(x)$ is not formally defined for $x > 1$. In practice this is often not too important (because PDFs drop rapidly towards $x = 1$), but we shall include two options: one that uses effective zero-points beyond $x = 1$ and one that ensures that the interpolation is based only on the physically valid domain of the PDFs.

[5] From the numerical point of view, it is advantageous to interpolate xq rather than q itself because the former is in general smoother.

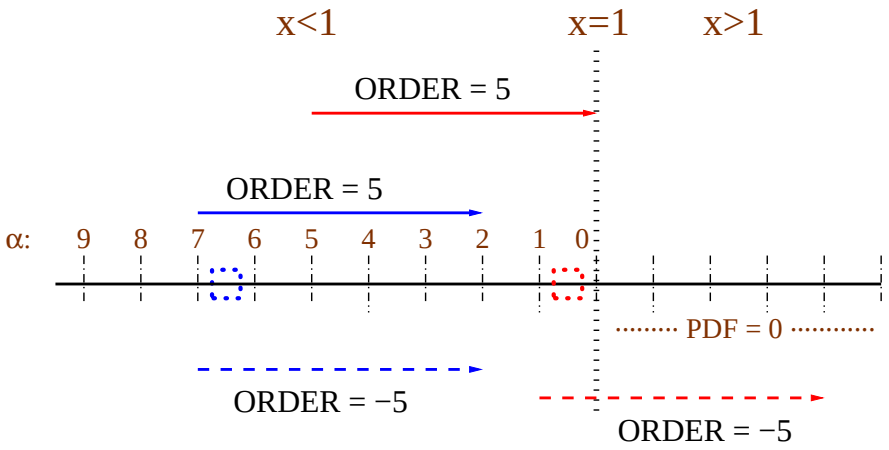

Figure 1: The different strategies (+ve and −ve order) for interpolation of the grid near $x = 1$. The dotted (blue, red) boxes indicate two regions in which we illustrate the interpolation of the PDF, while the (blue, red) lines with arrows indicate the corresponding range of grid points on which the interpolation is based.

These two choices are represented in Fig. 1. For an interpolation of order n (that is, which uses information from $n+1$ grid points), the option of using only points with $x \leq 1$ is denoted by `order` = `n`. This has the consequence that for $\beta \leq n$ we cannot write $P_{\alpha\beta} = \mathcal{P}_{\alpha-\beta}$, and so must explicitly store $\mathcal{O}(N_x n)$ distinct $P_{\alpha\beta}$ entries. The option of using artificial (zero-valued) points at $x > 1$ is denoted by `order` = $-$`n`, and does allow us to write $P_{\alpha\beta} = \mathcal{P}_{\alpha-\beta}$ (thus we store only $\mathcal{O}(N_x)$ entries), with $\mathcal{P}_{\alpha-\beta} = 0$ for $\alpha < \beta$.

Note that the piecewise interpolating polynomials that we use effectively imply a PDF that is not smooth at the grid points. In practice this is a small effect. We could, in principle, have enforced smoothness, for example by using splines. However analytical studies indicate suggest that for a given polynomial interpolation order this would actually reduce the accuracy of the convolutions. It would also complicate somewhat the internal bookkeeping during convolutions.

3.2 Evolution operators

The DGLAP evolution equation, eq. (3), is easily approximated in terms of its grid representation by

$$\frac{\partial q_\alpha(t)}{\partial t} = \frac{\alpha_s(t)}{2\pi} \sum_\beta P_{\alpha\beta}(t) q_\beta(t) \,, \tag{17}$$

where for a general value of α the sum over β extends over $\mathcal{O}(\alpha + |\texttt{order}|)$ points of the grid. Introducing $M_{\alpha\beta}(t_0) = \delta_{\alpha\beta}$ as an initial condition at some initial scale t_0, one can alternatively solve

$$\frac{\partial M_{\alpha\beta}(t)}{\partial t} = \frac{\alpha_s(t)}{2\pi} \sum_\gamma P_{\alpha\gamma}(t) M_{\gamma\beta}(t) \,. \tag{18}$$

Then the evolved parton distribution at the grid points is given by

$$q_\alpha(t) = \sum_\beta M_{\alpha\beta}(t) q_\beta(t_0)\,. \tag{19}$$

We refer to $M_{\alpha\beta}(t)$ as the evolution operator. From a practical point of view, we will solve eqs. (17) and (18) with higher order iterative Runge-Kutta methods, as described in section 6.2.1.

A further simplification occurs if one can rewrite the splitting functions P as translationally invariant objects, i.e. $P_{\alpha\beta} = \mathcal{P}_{\alpha-\beta}$. Then similarly one can rewrite $M_{\alpha\beta} = \mathcal{M}_{\alpha-\beta}$, and it is as simple to determine $M_{\alpha\beta}(t)$ as it is to determine the evolution of a single vector q_α, i.e. one just evolves a single column, $\beta = 0$, of $M_{\alpha\beta}(t)$.

4 Single-flavour grids and convolutions

HOPPET is written in Fortran 95 (F95). This has considerable advantages compared to F77, as will be seen in the discussion of the program, though it does lack a number of fully object-oriented features and this sometimes restricts the scope for expressiveness. Fortran 95 perhaps not being the best known language in the high-energy physics community, occasionally some indications will be give to help the reader with less-known language constructs, with further information in Appendix E.

All routines described in this section need access to the `convolution` module, which can either be obtained directly by adding a

```
use convolution
```

statement at the beginning of the relevant subprogram (before any `implicit none` or similar declarations). Alternatively, as with the rest of the routines described in this documentation, it can be accessed indirectly through the `hoppet_v1` module

```
use hoppet_v1
```

Unless you are delving into the innards of HOPPET, the latter is more convenient since it provides access to everything you are likely to need. Some of the more internal HOPPET routines and functions have however been left out of the `hoppet_v1` module, in order to reduce the likelihood of conflicts with objects in the user's namespace.

4.1 Grid definitions (`grid_def`)

The grid (in y) is the central element of the PDF evolution. Information concerning the grid is stored in a derived type `grid_def`:

```
type(grid_def) :: grid
call InitGridDef(grid,dy=0.1_dp,ymax=10.0_dp,order=5)
```

This initialises a grid between $x = 1$ and down to $x = e^{-\texttt{ymax}}$, with uniform grid spacing in $y = \ln 1/x$ of `dy=0.1`, with a grid that uses order 5 interpolation with only $x \leq 1$ points.

The user can modify this choice to better suit his/her particular needs, as explained in section 3.1. One notes the use of keyword arguments — the keywords are not mandatory in this case, but have been included to improve the legibility. Having defined a grid, the user need not worry about the details of the grid representation.

In line with the convention set out in the Fortran 90 edition of Numerical Recipes [37] we shall use `_dp` to indicate that floating-point numbers are in double precision, and `real(dp)` to declare double precision variables. The integer parameter `dp` is defined in the `module types` (and available indirectly through `module hoppet_v1`).

It is often useful to have multiple grids, with coarser coverage at small x and finer coverage at high x, to improve the precision of the convolution[6] at large-x without introducing an unnecessarily large density of points at small-x. To support this option, we can first define an array of sub-grids, and then use them to initialise a combined grid as follows:

```
type(grid_def) :: grid, subgrids(3)

! define the various sub-grids
call InitGridDef(subgrids(1),dy=0.30_dp, ymax=10.0_dp, order=5)
call InitGridDef(subgrids(2),dy=0.10_dp, ymax= 2.0_dp, order=5)
call InitGridDef(subgrids(3),dy=0.03333_dp, ymax= 0.6_dp, order=5)
             ! Smaller dy at small ymax / large xmin

! put them together into a single combined grid
call InitGridDef(grid, subgrids, locked=.true.)
```

When combining them, the `locked=.true.` option has been specified, which ensures that after any convolution, information from the finer grids is propagated into the coarser ones. This places some requirements on the grid spacings, notably that a coarse grid have a spacing that is a multiple of that of the next finest grid. If the requirements are not satisfied by the `subgrids` that have been provided, then new similar, but more suitable subgrids are automatically generated. When nested sub-grids are put together, the values of the x points of the combined grid will in general not be ordered.

Note that the two kinds of invocation of `InitGridDef` actually correspond to different (overloaded) subroutines. The Fortran 95 compiler automatically selects the correct one on the basis of the types of arguments passed.

Though only grids that are uniform in y have been implemented (and the option of a simultaneous combination of them), nearly all of the description that follows and all code

[6] The reason that denser grids are required at large-x is that if a typical parton distributions goes as

$$\lim_{x\to 1} q(x,Q^2) \sim (1-x)^m \ , \tag{20}$$

then its logarithmic derivative with respect to x is divergent,

$$\lim_{x\to 1} \frac{\partial \ln q(x,Q^2)}{\partial \ln x} = \lim_{x\to 1} \frac{-mx}{1-x} \to -\infty \ , \tag{21}$$

and therefore to maintain the relative accuracy of the evolution, grids with denser coverage at large-x are required.

outside the `convolution` module are independent of this detail, the only exception being certain statements about timings. Therefore were there to be a strong motivation for an alternative, non-uniform grid, it would suffice to modify the `convolution` module, while the rest of the library (and its interfaces) would remain unchanged.

4.2 x-space functions

Normal x–space functions (such as PDFs) are held in double precision arrays, which are to be allocated as follows

```
real(dp), pointer :: xgluon(:)
call AllocGridQuant(grid,xgluon)
```

Note that for this to work, `xgluon(:)` should be a `pointer`, and not just have the `allocatable` attribute. To deallocate a grid quantity, one may safely use the F95 `deallocate` command.

Since `xgluon(:)` is just an array, it carries no information about the `grid`. Therefore to set and access its value, one must always provide the information about the `grid`. This is not entirely satisfactory, and is one of the drawbacks of the use of F95.

There are a number of ways of setting a grid quantity. Suppose for example that we have a function

```
function example_xgluon(y)
  use types                !! defines "dp" (double precision) kind
  implicit none
  real(dp), intent(in)  :: y
  real(dp) :: x
  x = exp(-y)
  example_xgluon = 1.7_dp * x**(-0.1_dp) * (1-x)**5 !! returns xg(x)
end function example_xgluon
```

which returns the gluon momentum density $xg(x)$ (cf. section 3.1). Then we can call

```
call InitGridQuant(grid,xgluon,example_xgluon)
```

to initialise `xgluon` with a representation of the return value from the function `example_xgluon`. Alternative methods for initialising grid quantities are described in Appendix C.

To then access the gluon at a given value of $y = \ln 1/x$, one proceeds as follows

```
real(dp) :: y, xgluon_at_y
...
y = 5.0_dp
xgluon_at_y = EvalGridQuant(grid,xgluon,y) !! again this returns xg(x), x=exp(-y)
```

Note that again we have to supply the `grid` argument to `EvalGridQuant` because the `xgluon` array itself carries no information about the grid (other than its size).

A less efficient, but perhaps more 'object-oriented' way of accessing the gluon is via the notation

```
xgluon_at_y = xgluon .aty. (y.with.grid)
```

There also exists an `.atx.` operator for evaluating the PDF at a given x value. Many of these procedures and operators are overloaded so as to work with higher-dimensional arrays of grid quantities, for example a multi-flavour PDF array `pdf(:,:)`. The first index will always correspond to the representation on the grid, while the second index would here indicate the flavour.

Note that arithmetic operators all have higher precedence than library-defined operators such as `.aty.`; accordingly some ways of writing things are more efficient than others:

```
xgluon_at_y_times_2 = 2 * xgluon .aty. (y.with.grid)   ! very inefficient
xgluon_at_y_times_2 = 2 * (xgluon .aty. (y.with.grid)) ! fairly efficient
xgluon_at_y_times_2 = 2 * EvalGridQuant(grid,xgluon,y) ! most efficient
```

In the first case the whole of the array `xgluon` is multiplied by 2, and then the result is evaluated at y, whereas in the second and third cases only the result of the gluon at y is multiplied by 2.

4.3 Grid convolution operators

While it is relatively straightforward internally to represent a grid-quantity (e.g. a PDF) as an array, for convolution operators it is generally useful to have certain extra information. Accordingly a derived type has been defined to hold a convolution operator, and routines are provided for allocation and initialisation of splitting functions. The following example describes how the gg LO splitting function would be used to initialise the corresponding convolution operator:

```
type(grid_conv) :: xPgg
call AllocGridConv(grid,xPgg)
call InitGridConv(grid,xPgg, xPgg_func)
```

where the P_{gg} splitting function is provided in the form of the function `xPgg_func`. Note that this function must return $xP_{gg}(x)$:

```
! returns various components of exp(-y) P_gg (exp(-y))
real(dp) function xPgg_func(y)
  use types
  use convolution_communicator ! provides cc_piece, and cc_REAL,...
  use qcd                      ! provides CA, TR, nf, ...
  implicit none
  real(dp), intent(in) :: y
  real(dp)             :: x

  x = exp(-y); xPgg_func = zero
  if (cc_piece == cc_DELTA) then   ! Delta function term
     xPgg_func = (11*CA - 4*nf*TR)/6.0_dp
  else
     if (cc_piece == cc_REAL .or. cc_piece == cc_REALVIRT) &
         & xPgg_func = 2*CA*(x/(one-x) + (one-x)/x + x*(one-x))
     if (cc_piece == cc_VIRT .or. cc_piece == cc_REALVIRT) &
         & xPgg_func = xPgg_func - 2*CA*one/(one-x)
     xPgg_func = xPgg_func * x     ! remember to return x * Pgg
```

```
  end if
end function xPgg_func
```

To address the issue that convolution operators can involve plus-distributions and delta functions, the module `convolution_communicator` contains a variable `cc_piece` which indicates which part of the splitting function is to be returned — the real, virtual, real + virtual, or δ-function pieces.

The initialisation of a `grid_conv` object uses adaptive Gaussian integration (a variant of CERNLIB's `dgauss`) to calculate the convolution of the splitting function with trial weight functions. The default accuracy for these integrations is 10^{-7}. It can be modified to value `eps` with the following subroutine call

```
call SetDefaultConvolutionEps(eps)
```

which is to be made before creating the `grid_def` object. Alternatively, an optional `eps` argument can be included in the call to `InitGridDef` as follows:

```
type(grid_def) :: grid
real(dp) :: eps
[ ... set eps ... ]
call InitGridDef(grid,dy=0.1_dp,ymax=10.0_dp,order=3,eps)
```

Note that `eps` is just one of the parameters affecting the final accuracy of convolutions. In practice (unless going to extremely high accuracies) the grid spacing and interpolation scheme are more critical.

Having allocated and initialised a `xPgg` splitting function, we can go on to use it. For example:

```
real(dp), pointer :: xPgg_x_xgluon(:)
...
call AllocGridQuant(grid,xPgg_x_xgluon) !! Allocate memory for result of convolution
xPgg_x_xgluon = xPgg .conv. xgluon      !! Convolution of xPgg with xgluon
```

Since the return value of `xPgg .conv. xgluon` is just an F95 array, one can also write more complex expressions. Supposing we had defined also a `xPgq` splitting function and a singlet quark distribution `xquark`, as well as `as2pi` $= \alpha_s/2\pi$, then to first order in α_s we could write the gluon evolution through a step `dt` in $\ln Q^2$ as

```
xgluon = xgluon + (as2pi*dt) * ((xPgg .conv. xgluon) + (xPgq .conv. xquark))
```

Note that like `.aty.`, `.conv.` has a low precedence, so the use of brackets is important to ensure that the above expressions are sensible. Alternatively, the issues of precedence can be addressed by using `*` (also defined as convolution when it appears between a splitting function and a PDF) instead of `.conv.`:

```
xgluon = xgluon + (as2pi*dt) * (xPgg*xgluon + xPgq*xquark)
```

Note that, for brevity, from now on we will drop the explicit use of x in front of names PDF and convolution operator variables.

4.3.1 Other operations on grid_conv objects

It is marginally less transparent to manipulate grid_conv types than PDF distributions, but still fairly simple:

```
call AllocGridConv(grid,Pab)         ! Pab memory allocated
call InitGridConv(grid,Pab)          ! Pab = 0             (opt.alloc)
call InitGridConv(Pab,Pcd[,factor])  ! Pab = Pcd [*factor] (opt.alloc)
call InitGridConv(grid,Pab,function) ! Pab = function      (opt.alloc)

call SetToZero(Pab)                  ! Pab = 0
call Multiply (Pab,factor)           ! Pab = Pab * factor
call AddWithCoeff(Pab,Pcd[,coeff])   ! Pab = Pab + Pcd [*coeff]
call AddWithCoeff(Pab,function)      ! Pab = Pab + function

call SetToConvolution(Pab,Pac,Pcb)   ! Pab = Pac.conv.Pcb  (opt.alloc)
call SetToConvolution(P(:,:),Pa(:,:),Pb(:,:))            ! (opt.alloc)
                                     ! P(:,:) = matmul(Pa(:,:),Pb(:,:))
call SetToCommutator(P(:,:),Pa(:,:),Pb(:,:))             ! (opt.alloc)
                                     ! P(:,:) = matmul(Pa(:,:),Pb(:,:))
                                     !         -matmul(Pb(:,:),Pa(:,:))

call Delete(Pab)                     ! Pab memory freed
```

Routines labelled "(opt.alloc.)" allocate the memory for the grid_conv object if the memory has not already been allocated. (If it has already been allocated it is assumed to correspond to the same grid as any other grid_conv objects in the same subroutine call). Some calls require that one specify the grid definition being used (grid), because otherwise there is no way for the subroutine to deduce which grid is being used.

If repeatedly creating a grid_conv object for temporary use, it is important to remember to Delete it afterwards, so as to avoid memory leaks.

Nearly all the routines are partially overloaded so as to be able to deal with one and two-dimensional arrays of grid_conv objects as well. The exceptions are those that initialise the grid_conv object from a function (arrays of functions do not exist), as well as the convolution routines (for which the extension to arrays might be considered non-obvious) and the commutation routine which only has sense for matrices of grid_conv objects.

4.3.2 Derived grid_conv objects

Sometimes it can be cumbersome to manipulate the grid_conv objects directly, for example when trying to create a grid_conv that represents not a fixed order splitting function, but the resummed evolution from one scale to another. For such situations the following approach can be used

```
real(dp), pointer :: probes(:,:)
type(grid_conv)   :: Pqg, Pgq, Presult
integer           :: i

call GetDerivedProbes(grid,probes) ! get a set of 'probes'
```

```
do i = 1, size(probes,dim=2)       ! carry out operations on each of the probes
  probes(:,i) = Pqg*(Pgq*probes(:,i)) - Pgq*(Pqg*probes(:,i))
end do
call AllocGridConv(grid,Presult)
call SetDerivedConv(Presult,probes) ! Presult = [Pqg,Pgq]
```

Here `GetDerivedProbes` allocates and sets up an array of probe parton distributions. Since a single-flavour parton distribution is a one-dimensional array of `real(dp)`, the array of probes is a two-dimensional array of `real(dp)`, the second dimension corresponding to the index of the probe. One then carries out whatever operations one wishes on each of the probes. Finally with the call to `SetDerivedConv`, one can reconstruct a `grid_conv` object that corresponds to the set of operations just carried out

Some comments about memory allocation: the probes are automatically allocated and deallocated; in contrast the call to `SetDerivedConv(Presult,probes)` knows nothing about the grid, so `Presult` must have been explicitly allocated for a specific grid beforehand.

A note of caution: when one's grid is made of nested subgrids with the locking option set to `.true.`, after a convolution of a `grid_def` object with a parton distribution, the coarser grids for the parton distribution are supplemented with more accurate information from the finer grids. When carrying out multiple convolutions, this happens after each convolution. There is no way to emulate this with a single `grid_def` object, and the locking would actually confuse the reconstruction of resulting `grid_def` object. So when the user requests the probes, locking is temporarily turned off globally and then reestablished after the derived `grid_object` has been constructed. Among other things this means that acting with a derived `grid_object` will not be fully equivalent to carrying out the individual operations separately. In particular the accuracy may be slightly lower (whatever is lost due to the absence of intermediate locking).

4.4 Truncated moments

There are various contexts in which it is useful to evaluate moments of PDFs, notably in checking sum rules. Since HOPPET doesn't store PDFs over the full range of x, one cannot calculate full moments, but only truncated moments. Truncated moments are calculated via the function

```
TruncatedMoment(grid, xparton, N [,ymax])
```

with the convention that the result M_N is

$$M_N[\texttt{xparton}] \equiv \int_0^{y_{\max}} dy\, e^{-yN}\, \texttt{xparton.aty.(}y\texttt{.with.grid)}\,. \tag{22}$$

The moment index N must be a double precision number, and if the argument `ymax` is not supplied, the integral extends to largest y value present in the grid, `grid%ymax`. The integral is evaluated using adaptive Gaussian integration with the same default precision that was set for the determination of grid convolution operators. The use of adaptive

Gaussian integration implies that the evaluation of moments is not optimally efficient for repeated use of the same moment index with different PDFs.[7]

As as example, the calculation of the momentum contained in the gluon would proceed as follows:

```
real(dp), pointer :: xgluon(:)
real(dp)          :: N, gluon_momentum
! [ ... other vars, allocation, intialisation ...]
N = 1.0_dp
gluon_momentum = TruncatedMoment(grid, N, xgluon)
```

For a further example, see also section 7.2. Note that for `ymax` $= 12$ (a value used in many of the example programs), the truncation can underestimate the sum rules by a fraction of a percent. This underestimation is largest at higher values of Q^2, due to the perturbative growth of PDFs at small-x at large-Q^2.

4.5 Parton Luminosities

The parton luminosity function $\mathcal{L}(\tau)$ obtained from two parton distributions $xq_i(x)$ and $xq_j(x)$, defined as

$$\mathcal{L}_{ij}(\tau) = \int_\tau^1 \frac{dx}{x}\, xq_i(x)\, \frac{\tau}{x} q_j\left(\frac{\tau}{x}\right) = \tau \int_\tau^1 \frac{dx}{x} q_i(x)\, q_j\left(\frac{\tau}{x}\right) \tag{23}$$

is a quantity that enters evaluations of many total cross sections at hadron-hadron colliders: e.g. the LO cross section for producing a boson with mass M through fusion of partons of types i and j is proportional to $\mathcal{L}_{ij}(M^2/s)$, where s is the squared centre of mass energy of the hadron-hadron collision.

The parton luminosity function is represented as a grid quantity

```
real(dp), pointer :: lumi(:), xqi(:), xqj(:)
! [... lumi, xqi and xqj should all be allocated as standard grid quantities]
lumi = PartonLuminosity(grid, xqi, xqj)
```

and all standard operations on grid quantities can be used with it, for example convolutions with splitting functions, evaluations at particular values of y or x, and so forth.

Parton luminosities are a new feature of the 1.1.5 release of HOPPET and should be considered "beta" functionality, valid only for PDFs that vanish smoothly for $x \to 1$, the same assumption that enters the use of negative `order` values (cf. Fig. 1). In particular, for grids with a single spacing, they are evaluated through the sum

$$\mathcal{L}_{ij,\alpha} = \mathtt{dy} \times \sum_{\beta=0}^{\alpha} q_{i,\beta}\, q_{j,\alpha-\beta}\,. \tag{24}$$

At first sight, this might appear to be a low-order integration formula. However one can show that if one superposes multiple higher-order integration formulae, each shifted by one

[7] Should a user have an application in which the repeated evaluation of truncated moments is the main time-consuming step, then they are advised to contact the authors for further advice.

grid unit, then one obtains Eq. (24) as long as the integrand vanishes sufficiently smoothly at its upper and lower limits, as is the case with physical PDFs. When using multiple locked grid spacings, say a coarse and a fine grid, $\texttt{dy}_1 > \texttt{dy}_2$, $\texttt{ymax}_1 > \texttt{ymax}_2$, then for y up to $2\texttt{ymax}_2$ the luminosity sum is evaluated using the fine grid spacing, taking information directly from the fine grid for the PDFs up to $\texttt{ymax}_2$, plus interpolation from the coarse grid onto the fine grid for points up to $2\texttt{ymax}_2$ (the interpolation order is that of the coarse grid). This procedure has been found to give accuracies comparable to those obtained for convolutions. Note that it may be modified in future versions of HOPPET.

5 Multi-flavour grids and convolutions

The discussion in the previous section about how to represent functions and associated convolutions in a general $x-$space grid holds for any kind of problem involving convolutions, even if the examples were given in the context of DGLAP evolution. In this section we shall examine the tools made available specifically to address the DGLAP evolution problem.

5.1 Full-flavour PDFs and flavour representations

The routines described in this section are available from the `pdf_general` and `pdf_representation` modules, or via the `hoppet_v1` general module.

Full flavour PDFs sets are just like single flavour PDFs except that they have an extra dimension. They are represented by arrays, and if you want HOPPET to deal with allocation for you, they should be pointer arrays. One can allocate a single PDF (two dimensional `real(dp)` array) or an array of PDFs (three-dimensional `real(dp)` array)

```
real(dp), pointer :: PDF(:,:), PDFarray(:,:,:)
call AllocPDF(grid,PDF)            ! allocates PDF(0:,-6:7)
call AllocPDF(grid,PDFarray,0,10)  ! allocates PDFarray(0:,-6:7,0:10)
```

The first dimension corresponds to the grid in y; the second dimension corresponds to the flavour index. Its lower bound is -6, as one would expect.

What takes a bit more getting used to is that its upper bound is **7**. The reason is as follows: the flavour information can be represented in different ways, for example each flavour separately, or alternatively as singlet and non-singlet combinations. In practice both are used inside the program and it is useful for a PDF distribution to have information about the representation, and this is stored in `PDF(:,7)`[8].

5.1.1 Human representation.

When a PDF is allocated it is automatically labelled as being in the `human` representation, described in section 2. Constants with names like `iflv_bbar`, `iflv_g`, `iflv_b`, are defined

[8] In the current release of HOPPET, in particular, for PDFs in the `human` representation one has `PDF(:,7)=0`, while for PDFs in the `evln` representation, the information on the active number of flavours is encoded as `nf = (abs(q(2,7))+abs(q(3,7)))/(abs(q(0,7))+abs(q(1,7)))`, so that it is conserved under linear operations. However the details of the encoding may evolve in future versions of the program.

in `module pdf_representation`, to facilitate symbolic access to the different flavours.

If you are creating a PDF as an automatic array (one whose bounds are decided not by the allocation routine, but on the fly), for example in a function that returns a PDF, then you should label it yourself as being in the `human` representation, either with the `LabelPdfAsHuman(pdf)` subroutine call, or by setting `pdf(:,7)` to zero:

```
module pdf_initial_condition
  use hoppet_v1; implicit none
contains
  function unpolarized_dummy_pdf(xvals) result(pdf)
    real(dp), intent(in) :: xvals(:)
    real(dp)             :: pdf(size(xvals),-6:7)

    ! clean method for labelling a PDF as being in the human representation
    call LabelPdfAsHuman(pdf)
    ! Alternatively, by setting everything to zero
    ! (notably pdf(:,7)), the PDF representation
    ! is automatically set to be human
    pdf(:,:) = 0

    ! iflv_g is pre-defined integer parameter (=0) for symbolic ref. to gluon
    pdf(:,iflv_g) = 1.7_dp * xvals**(-0.1_dp) * (1-xvals)**5 ! Returns x*g(x)
    [... set other flavours here ...]
  end function unpolarized_dummy_pdf
end module pdf_initial_condition
```

The function has been placed in a module so as to provide an easy way for a calling routine to have access to its interface (this is needed for the dimension of `xvals` to be correctly passed). Writing a function such as that above is probably the easiest way of initialising a PDF:

```
  use hoppet_v1; use pdf_initial_condition; implicit none
  type(grid_def)    :: grid
  real(dp), pointer :: pdf(:,:)
  [...]
  call AllocPDF(grid,pdf)
  pdf = unpolarized_dummy_pdf(xValues(grid))
```

There exist a number of other options, which can be found by browsing through `src/pdf_general.f90`. Of these a sometimes handy one is

```
  call AllocPDF(grid,pdf)
  call InitPDF_LHAPDF(grid, pdf, LHAsub, Q)
```

where `LHAsub` is the name of a subroutine with the same interface as LHAPDF's `evolvePDF` [20]:

```
  subroutine LHAsub(x,Q,res)
    use types; implicit none
    real(dp), intent(in)  :: x,Q
    real(dp), intent(out) :: res(-6:6) ! on output contains flavours -6:6 at x,Q
    [...]                              ! Note that it should return momentum densities
  end subroutine LHAsub
```

Note that `LHAsub` should return momentum densities, as happens with the LHAPDF routines [20].

Having initialised a PDF, to then extract it at a given y value, one can either examine a particular flavour using the methods described in section 4.2

```
  real(dp) :: y, gluon_at_y
  gluon_at_y = pdf(:,iflv_g) .aty. (y.with.grid)
  ! OR
 gluon_at_y = EvalGridQuant(grid,pdf(:,iflv_g),y)
```

or one can extract all flavours simultaneously

```
  real(dp) :: pdf_at_y(-6:6)
  pdf_at_y = pdf(:,-6:6) .aty. (y.with.grid)
  ! OR
  pdf_at_y =  EvalGridQuant(grid,pdf(:,-6:6),y)
```

with the latter being more efficient if one needs to extract all flavours simultaneously. Note that here we have explicitly specified the flavours, `-6:6`, that we want.[9]

5.1.2 Evolution representation

For the purpose of carrying out convolutions, the `human` representation is not very advantageous because the splitting matrix in flavour space is quite complicated. Accordingly HOPPET uses a different representation of the flavour internally when carrying out convolution of splitting matrices with PDFs. For most purposes the user need not be aware of this. The two exceptions are when a user plans to create derived splitting matrices (being careless about the flavour representation will lead to mistakes) or wishes to carry out repeated convolutions for a fixed n_f value (appropriate manual changes of the flavour representation can speed things up).

The splitting matrix can be simplified considerably by switching to a different flavour representation, as can be seen in Table 1. When carrying out a convolution, the only non-diagonal part is the block containing indices $0, 1$. This representation is referred to as the `evln` representation. Whereas the `human` representation is n_f-independent, the `evln` depends on n_f through the Σ and $q_{\rm NS}^V$ entries and the fact that flavours beyond n_f are left in the human representation (since they are inactive for evolution with n_f flavours).

To take a PDF in the `human` representation and make a copy in an `evln` representation, one uses the `CopyHumanPdfToEvln` routine

```
  real(dp), pointer :: pdf_human(:,:), pdf_evln(:,:)
  integer           :: nf_lcl ! NB: nf would conflict with global variable
  [... setting up pdf_human, nf_lcl, etc. ...]
  call AllocPDF(grid,pdf_evln)  ! or it might be an automatic array
  call CopyHumanPdfToEvln(nf_lcl, pdf_human, pdf_evln) ! From human to evolution representation
```

where one specifies the n_f value for the `evln` representation. One can go in the opposite direction with

[9] If instead we had said `pdf(:,:)` the result would have corresponded to a slice of flavours `-6:7`, i.e. including an interpolation of the representation labelling information, which would be meaningless.

```
call CopyEvlnPdfToHuman(nf_lcl, pdf_evln, pdf_human)
```

At any time one can check which is the representation of a given PDF using the `GetPdfRep` function,

```
integer nf_rep
real(dp), pointer :: pdf(:,:)

[... set up pdf, ...]
nf_rep = GetPdfRep(pdf)
```

which returns the number of active flavours if the PDF is in the `evln` representation, or a negative integer if the PDF is in the `human` representation.

5.2 Splitting function matrices

Splitting function matrices and their actions on PDFs are defined in `module dglap_objects` (accessible as usual from `module hoppet_v1`). They have type `split_mat`. Below we shall discuss routines for creating specific predefined DGLAP splitting matrices, but for now we consider a general splitting matrix.

The allocation of `split_mat` objects,

```
type(split_mat) :: P
integer         :: nf_lcl
call AllocSplitMat(grid, P, nf_lcl)
```

is similar to that for `grid_conv` objects. The crucial difference is that one must supply a value for n_f, so that when the splitting matrix acts on a PDF it knows which flavours are decoupled. From the point of view of subsequent initialisation a `split_mat` object just consists of a set of splitting functions. If need be, they can be initialised by hand, for example

```
call InitGridConv(grid,P%qq      , P_function_qq      )
call InitGridConv(grid,P%qg      , P_function_qg      )
call InitGridConv(grid,P%gq      , P_function_gq      )
call InitGridConv(grid,P%gg      , P_function_gg      )
call InitGridConv(grid,P%NS_plus , P_function_NS_plus )
call InitGridConv(grid,P%NS_minus, P_function_NS_minus)
call InitGridConv(grid,P%NS_V    , P_function_NS_V    )
```

One can then write

```
real(dp), pointer :: q(:,:), delta_q(:,:)
[... allocations, etc. ...]
delta_q = P .conv. q
! OR
delta_q = P * q
```

and `delta_q` will have the following components

$$\begin{aligned}
\begin{pmatrix} \delta\Sigma \\ \delta g \end{pmatrix} &= \begin{pmatrix} \texttt{P\%qq} & \texttt{P\%qg} \\ \texttt{P\%gq} & \texttt{P\%gg} \end{pmatrix} \otimes \begin{pmatrix} \Sigma \\ g \end{pmatrix} \\
\delta q^{+}_{\mathrm{NS},i} &= \texttt{P\%NS_plus} \otimes q^{+}_{\mathrm{NS},i} \\
\delta q^{-}_{\mathrm{NS},i} &= \texttt{P\%NS_minus} \otimes q^{-}_{\mathrm{NS},i} \\
\delta q^{V}_{\mathrm{NS}} &= \texttt{P\%NS_V} \otimes q^{V}_{\mathrm{NS}}
\end{aligned} \tag{25}$$

We have written the result in terms of components in the `evln` representation (and this is the representation used for the actual convolutions). When a convolution with a PDF in `human` representation is carried out, the program automatically copies the PDF to the `evln` representation, carries out the convolution and converts the result back to the `human` representation. The cost of changing a representation is $\mathcal{O}(N_x)$, whereas the convolution is $\mathcal{O}(N_x^2)$, so in principle the former is negligible. In practice, especially when aiming for high speed at low N_x, the change of representation can imply a significant cost. In such cases, if multiple convolutions are to be carried out, it may be advantageous to manually change into the appropriate `evln` representation, carry out all the convolutions and then change back manually to the `human` representation at the end, see section 5.1.

As for `grid_conv` objects, a variety of routines have been implemented to help manipulate splitting matrices:

```
type(split_mat) :: PA, PB, PC
real(dp) :: factor

call InitSplitMat(PA,PB[,factor])   ! PA = PB [*factor]   (opt.alloc)

call SetToZero(PA)                  ! PA = 0
call Multiply(PA,factor)            ! PA = PA * factor
call AddWithCoeff(PA,PB[,factor])   ! PA = PA + PB [*factor]

call SetToConvolution(PA,PB,PC)     ! PA = PB*PC          (opt.alloc)
call SetToCommutator(PA,PB,PC)      ! PA = PB*PC-PC*PB    (opt.alloc)

call Delete(split_mat)              ! PA's memory freed
```

5.2.1 Derived splitting matrices

As with `grid_conv` objects, HOPPET provides means to construct a `split_mat` object that corresponds to an arbitrary series of `split_mat` operations, as long as they all involve the same value of n_f. One proceeds in a very similar way as in section 4.3.2,

```
real(dp), pointer :: probes(:,:,:)
type(split_mat)  :: PA, PB, Pcomm
integer          :: i

[...set nf_lcl,...]
```

```
call GetDerivedSplitMatProbes(grid,nf_lcl,probes) ! get the probes
do i = 1, size(probes,dim=3)               ! carry out operations on each probe
  probes(:,:,i) = PA*(PB*probes(:,:,i)) - PB*(PA*probes(:,:,i))
end do
call AllocSplitMat(grid,Pcomm,nf_lcl)      ! provide nf info in initialisation
call SetDerivedConv(Pcomm,probes)          ! Presult = [Pqg,Pgq]
```

Note that we need to provide the number of active quark flavours to `GetDerivedSplitMatProbes`. As in section 4.3.2, we first need to set up some 'probe' PDFs (note the extra dimension compared to earlier, since we also have flavour information; the probe index always corresponds to the last dimension); then we act on those probes; finally we allocate the splitting matrix, and set its contents based on the probes, which are then automatically deallocated.

5.3 The DGLAP convolution components

5.3.1 QCD constants

The splitting functions that we set up will depend on various QCD constants (n_f, colour factors), so it is useful to here to summarise how they are dealt with within the program.

The treatment of the QCD constants is *not* object oriented. There is a module (`qcd`) that provides access to commonly used constants in QCD:

```
real(dp) :: ca, cf, tr, nf
integer  :: nf_int

real(dp) :: beta0, beta1, beta2
[ ... ]
```

Note that `nf` is in double precision — if you want the integer value of n_f, use `nf_int`.

To set the value of n_f, call

```
integer :: nf_lcl
call qcd_SetNf(nf_lcl)
```

where we have used the local variable `nf_lcl` to avoid conflicting with the `nf` variable provided by the `qcd` module. Whatever you do, do not simply modify the value of the `nf` variable by hand — when you call `qcd_SetNf` it adjusts a whole set of other constants (e.g. the β function coefficients) appropriately.

There are situations in which it's of interest to vary the other colour factors of QCD, for example, if these colour factors are to be determined from a fit to deep-inelastic scattering experimental data. For that purpose, use

```
real(dp) :: ca_lcl, cf_lcl, tr_lcl
call qcd_SetGroup(ca_lcl, cf_lcl, tr_lcl)
```

Again all other constants in the `qcd` module will be adjusted. A word of caution: the NNLO splitting functions actually depend on a colour structure that goes beyond the usual C_A, C_F and T_R, namely $d_{abc}d^{abc}$, which in the present version of HOPPET is hard-wired to its default QCD value.

5.3.2 DGLAP splitting matrices

The module `dglap_objects` includes a number of routines for providing access to the `split_mat` objects corresponding to DGLAP splitting functions

```
type(split_mat) :: P_LO, P_NLO, P_NNLO
type(split_mat) :: Pp_LO, Pp_NLO        ! polarised

! MSbar unpolarised case
call InitSplitMatLO  (grid, P_LO)
call InitSplitMatNLO (grid, P_NLO)
call InitSplitMatNNLO(grid, P_NNLO)

! the MSbar polarised case...
call InitSplitMatPolLO (grid, Pp_LO)
call InitSplitMatPolNLO(grid, Pp_NLO)
```

In each case the splitting function is set up for the n_f and colour-factor values that are current in the `qcd` module, as set with the `qcd_SetNf` and `qcd_SetGroup` subroutine calls. If one subsequently resets the n_f or colour factor values, the `split_mat` objects continue to correspond to the n_f and colour factor values for which they were initially calculated. With the above subroutines for initialising DGLAP splitting functions, the normalisation is as given in eq. (4).

When carrying out DGLAP evolution it is most efficient to first sum the splitting matrices and then carry out the convolution,

```
type(split_mat)   :: P_sum
real(dp), pointer :: q(:,:), dq(:,:)
[ ... ]
call InitSplitMat(P_sum, P_LO)            ! P_sum = P_LO
call AddWithCoeff(P_sum, P_NLO, as2pi)    ! P_sum = P_sum + as2pi * P_NLO
dq = (as2pi * dt) * (P_sum .conv. q)      ! Step dt in evolution
call Delete(P_sum)                        ! Memory freed
```

This is because convolutions take a time $\mathcal{O}(N^2)$, where N is the number of points in the grid, whereas additions and multiplications take a time $\mathcal{O}(N)$. Note the use of brackets in the line setting `dq`: all scalar factors are first multiplied together ($\mathcal{O}(1)$) so that we only have one multiplication of a PDF ($\mathcal{O}(N_x)$). Note also that we have chosen to include the `(as2pi * dt)` factor as multiplying the pdf, rather than the other option of multiplying P_sum, i.e.

```
call Multiply(P_sum, (as2pi * dt))
dq =  P_sum .conv. q
```

The result would have been identical, but splitting matrices with positive interpolation `order` essentially amount to an $\mathcal{O}(7 \times \texttt{order} \times N)$ sized array, whereas the PDF is an $\mathcal{O}(13N)$ sized array and the for high positive orders that are sometimes used, it is cheaper to multiply the latter.

The default for the NNLO splitting functions are the interpolated expressions, which are very fast to evaluate. Other possibilities, like the exact splitting functions or a previous

set of approximated NNLO splitting functions which was used before the full calculation was available are described in Appendix D. Note that the QCD colour factors introduced in section 5.3.1 cannot be modified if the interpolated NNLO splitting functions are used, since these expressions use the default QCD values.

5.3.3 Mass threshold matrices

Still in the `dglap_objects` module, we have a type dedicated to crossing heavy quark mass thresholds.

```
type(grid_def)           :: grid
type(mass_threshold_mat) :: MTM_NNLO

call InitMTMNNLO(grid, MTM_NNLO)   ! MTM_NNLO is coeff of (as/2pi)**2
```

This is the coefficient of $(\alpha_s/2\pi)^2$ for the convolution matrix that accounts for crossing a heavy flavour threshold in $\overline{\text{MS}}$ factorisation scheme, at $\mu_F = m_h$, where m_h is the heavy-quark pole mass (or $\overline{\text{MS}}$ mass), as has been described in section 2. Since the corresponding NLO term is zero, the number of flavours in α_s is immaterial at NNLO.

The treatment of n_f in the `mass_threshold_mat` is very specific because at NNLO, the only order in the $\overline{\text{MS}}$ factorisation scheme at which it's non-zero and currently known, it is independent of n_f. Its action does of course however depend on n_f. Since, as for `split_mat` objects, we don't want the action of the `mass_threshold_mat` to depend on the availability of the current n_f information from the `qcd` module, instead we require that before using a `mass_threshold_mat`, you should explicitly indicate the number of flavours (defined as including the new heavy flavour). This is done using a call to the `SetNfMTM(MTM_NNLO,nf_incl_heavy)` subroutine. A similar function `SetMassSchemeMTM( MTM_NNLO, masses_are_MSbar)` can be used to specify whether the threshold is being crossed at a pole mass or $\overline{\text{MS}}$ mass. So for example to take a PDF in the effective theory with $n_f = 3$ active flavours `pdf_nf3`, and convert it to the corresponding PDF in the effective theory with $n_f = 4$ active flavours `pdf_nf4` at m_h^2 (pole mass), one uses code analogous to the following

```
real(dp) :: pdf_nf3(:,:), pdf_nf4(:,:)
logical  :: masses_are_MSbar = .false.
[ ... ]
call SetNfMTM(MTM, 4)
call SetMassSchemeMTM(MTM, masses_are_MSbar)
pdf_nf4 = pdf_nf3 + (as2pi)**2 * (MTM_NNLO.conv.pdf_nf3)
```

The convolution only works if the `pdf`'s are in the `human` representation and an error is given if this is not the case. Any heavy flavour (like for example intrinsic charm) present in `pdf_nf3` would be left in unchanged.

Note that the type `mass_threshold_mat` is not currently suitable for general changes of flavour-number. For example if you wish to carry out a change in the DIS scheme or at a scale $\mu_F \neq m_h$ then you have to combine a series of different convolutions (essentially correcting with the lower number of flavours to the $\overline{\text{MS}}$ factorisation scheme at $\mu_F = m_h$

before changing the number of flavours and then correcting back to the original scheme and scale using the higher number of flavours).

As for the NNLO splitting functions, the mass threshold corrections come in exact and parametrised variants. By default it is the latter that is used (provided by Vogt [38]). The cost of initialising with the exact variants of the mass thresholds is much lower than for the exact NNLO splitting functions (partly because there is no n_f dependence, partly because it is only one flavour component of the mass-threshold function that is complex enough to warrant parametrisation). The variant can be chosen by the user before initialising the `mass_threshold_mat` by making the following subroutine call:

```
integer :: threshold_variant
call dglap_Set_nnlo_nfthreshold(threshold_variant)
```

with the following variants defined (as integer parameters), again in the module `dglap_choices`:

```
nnlo_nfthreshold_exact
nnlo_nfthreshold_param                [default]
```

5.3.4 Putting it together: `dglap_holder`

The discussion so far in this subsection was intended to provide the reader with an overview of the different DGLAP components that have been implemented and of how they can be initialised individually. This is useful above all if the user needs to tune the program to some specific unusual application.

In practice, we foresee that most users will need just a standard DGLAP evolution framework, and so will prefer not need to manage all these components individually. Accordingly HOPPET provides a type, `dglap_holder` which holds all the components required for a given kind of evolution. To initialise all information for a fixed-flavour number evolution, one does as follows

```
use hoppet_v1
type(dglap_holder) :: dglap_h
integer            :: factscheme, nloop, nf_lcl

nloop      = 3                 ! NNLO
factscheme = factscheme_MSbar  ! or: factscheme_DIS; factscheme_PolMSbar
nf_lcl = 4
call qcd_SetNf(nf_lcl)         ! set the fixed number of flavours
! call qcd_SetGroup(...)       ! if you want different colour factors

! now do the initialisation
call InitDglapHolder(grid, dglap_h, factscheme, nloop)
```

The constants `factscheme_*` are defined in `module dglap_choices`. The corrections to the splitting functions to get the DIS scheme are implemented by carrying out appropriate convolutions of the $\overline{\rm MS}$ splitting and coefficient functions. Currently the DIS scheme is only implemented to NLO.[10] The polarised splitting functions are only currently known to

[10] Its NNLO implementation would actually be fairly straightforward given the parametrisation provided in [39], and may be performed in future releases of HOPPET.

NLO.

Initialisation can also be carried out with a single call for a range of different numbers of flavours:

```
integer :: nflo, nfhi
[...]
nflo = 3; nfhi = 6    ! [calls to qcd_SetNf handled automatically]
call InitDglapHolder(grid, dglap_h, factscheme, nloop, nflo, nfhi)
```

Mass thresholds are not currently correctly supported in the DIS scheme, even at NLO.

For all the above calls, at NNLO the choice of exact of parametrised splitting functions and mass thresholds is determined by the calls to `dglap_Set_nnlo_splitting` and `dglap_Set_nnlo_nfthreshold`, as described in sections 5.3.2 and 5.3.3 respectively. These calls must be made prior to the call to `InitDglapHolder`.

Having initialised a `dglap_holder` one has access to various components:

```
type dglap_holder
  type(split_mat), pointer :: allP(1:nloop, nflo:nfhi)  ! FFNS: nflo=nfhi=nf_lcl
  type(split_mat), pointer :: P_LO, P_NLO, P_NNLO
  type(mass_threshold_mat) :: MTM2
  logical                  :: MTM2_exists
  integer                  :: factscheme, nloop
  integer                  :: nf
  [ ... ]
end type dglap holder
```

Some just record information passed on initialisation, for example `factscheme` and `nloop`. Other parts are set up once and for all on initialisation, notably the `allP` matrix, which contains the 1-loop, 2-loop, etc. splitting matrices for the requested n_f range.

Yet other parts of the `dglap_holder` type depend on n_f. Before accessing these, one should first perform the following call:

```
call SetNfDglapHolder(dglap_h, nf_lcl)
```

This creates links:

```
dglap_h%P_LO   => dglap_h%allP(1,nf_lcl)
dglap_h%P_NLO  => dglap_h%allP(2,nf_lcl)
dglap_h%P_NNLO => dglap_h%allP(3,nf_lcl)
```

for convenient named access to the various splitting matrices, and it also sets the global (`qcd`) n_f value (via a call to `qcd_SetNf`) and where relevant updates the internal n_f value associated with `MTM2` (via a call to `SetNfMTM`).

As with other types that allocate memory for derived types, that memory can be freed via a call to the `Delete` subroutine,

```
call Delete(dglap_h)
```

6 DGLAP evolution

So far we have described all the tools that are required to perform DGLAP convolutions of PDFs. In this section we describe how the different ingredients are put together to perform the actual evolution.

6.1 Running coupling

Before carrying out any DGLAP evolutions, one first needs to set up a `running_coupling` object (defined in `module qcd_coupling`):

```
type(running_coupling) :: coupling
real(dp) :: alfas, Q, quark_masses(4:6), muMatch_mQuark
integer  :: nloop, fixnf

[... set parameters ...]
call InitRunningCoupling(coupling [, alfas] [, Q] [, nloop] [, fixnf]&
                       & [, quark_masses] [, masses_are_MSbar] [, muMatch_mQuark])
```

As can be seen, many of the arguments are optional. Their default values are as follows:

```
Q     = 91.2_dp
alfas = 0.118_dp  ! Value of coupling at scale Q

nloop = 2
fixnf = [.not. present]
                      !     charm,      bottom,   top
quark_masses(4:6) = (/ 1.414213563_dp, 4.5_dp, 175.0_dp /) ! Heavy quark pole masses
muMatch_mQuark    = 1.0_dp
```

The running coupling object is initialised so that at scale `Q` the coupling is equal to `alfas`. The running is carried out with the `nloop` β-function. If the `fixnf` argument is present, then the number of flavours is kept fixed at that value. Otherwise flavour thresholds are implemented at scales

```
muMatch_mQuark * quark_masses(4:6)
```

By default the quark masses are taken to be *pole* masses. This choice (and the particular default values) was inspired by the PDF evolution benchmark comparison [19] in which HOPPET results were compared to those of Vogt's moment-space code QCD-Pegasus [3]. If the user instead prefers to supply $\overline{\rm MS}$ masses, $m_h^{\overline{\rm MS}}(m_h^{\overline{\rm MS}})$, and have flavour thresholds implemented at the $\overline{\rm MS}$ masses, then she/he should specify `masses_are_MSbar = .true.`.

The default value of the QCD coupling is taken to be close to the world average at the time of initial release of HOPPET [40].

To access the coupling at some scale `Q` one uses the following function call:[11]

```
alfas = Value(coupling, Q [, fixnf])
```

[11] `RunningCoupling(...)` can be used as a synonym for `Value(...)`, which helps code readability when the `coupling` argument is left out (i.e. when using the global `coupling` object).

This is the value of the coupling as obtained from the Runge-Kutta solution of the `nloop` version of eq. (5) (the numerical solution is actually carried out for $1/\alpha_s$), together with the appropriate mass thresholds. For typical values of $\alpha_s(M_Z)$ the coupling is guaranteed to be reliably determined in the range 0.5 GeV $< Q <$ 10^{19} GeV. The values of the β function coefficients used in the evolution correspond to those obtained with the values of the QCD colour factors that were in vigour at the moment of initialisation of the coupling.

In the variable flavour-number case, the `fixnf` argument allows one to obtain the coupling for `fixnf` flavours even outside the natural range of scales for that number of flavours. This is only really intended to be used close to the natural range of scales, and can be slow if one goes far from that range (a warning message will be output). If one is interested in a coupling that (say) never has more than 5 active flavours, then rather than using the `fixnf` option in the `Value` subroutine, it is best to initialise the coupling with a fictitious large value for the top mass.

Often it is convenient to be able to enquire about the mass information embodied in a `running_coupling`. For example in the PDF evolution below, all information about the location of mass thresholds is obtained from the `running_coupling` type.

The quark mass for flavour `iflv` can be obtained with the call

```
quark_mass = QuarkMass(coupling, iflv)
```

This is the pole mass, unless the function `QuarkMassesAreMSbar` returns `.true.`, in which case it is an $\overline{\rm MS}$ mass. The range of scales, `Qlo` $< Q <$ `Qhi` for which `iflv` is the heaviest active flavour is obtained by the subroutine call

```
call QRangeAtNf(coupling, iflv, Qlo, Qhi [, muM_mQ])
```

The optional argument `muM_mQ` allows one to obtain the answer as if one had initialised the coupling with a different value of `muMatch_mQuark` than that actually used. One can also establish the number of active flavours, `nf_active`, at a given scale `Q` with the following function:

```
nf_active = NfAtQ(coupling, Q [, Qlo, Qhi] [, muM_mQ])
```

As well as returning the number of active flavours, it can also set `Qlo` and `Qhi`, which correspond to the range of scales in which the number of active flavours is unchanged. The optional `muM_mQ` argument has the same purpose as in the `QRangeAtNf` subroutine. The last of the enquiry functions allows one to obtain the range of number of flavours covered in this coupling, `nflo` $\leq n_f \leq$ `nfhi`:

```
call NfRange(coupling, nflo, nfhi)
```

Finally, as usual, once you no longer need a `running_coupling` object, you may free the memory associated with it using the `Delete` call:

```
call Delete(coupling)
```

6.2 DGLAP evolution

6.2.1 Direct evolution

We are now, at last, ready to evolve a multi-flavour PDF. This is done by breaking the evolution into steps, and for each one using a Runge-Kutta approximation for the solution of a first-order matrix differential equation. The steps are of uniform size in a variable u that satisfies the following approximate relation

$$\frac{du}{d\ln Q^2} \simeq \alpha_s(Q^2) \ . \tag{26}$$

For a 1-loop running coupling one has $u = (\ln\ln Q^2/\Lambda)/\beta_0$, which is the variable that appears in analytical solutions to the 1-loop DGLAP equation. The step size in u, du, can be set with the following call

```
real(dp) :: du = 0.1_dp ! or some smaller value
call SetDefaultEvolutionDu(du)
```

The error on the evolution from the finite step size should scale as $(\texttt{du})^4$. With the default value of $\texttt{du} = 0.1$, errors are typically somewhat smaller than 10^{-3} (see section 9 for the detailed benchmarks).

To actually carry out the evolution, one uses the following subroutine call:

```
type(dglap_holder)     :: dglap_h
type(running_coupling) :: coupling
real(dp), pointer      :: initial_pdf(:,:)
real(dp)               :: Q_init, Q_end
integer                :: nloop
integer                :: untie_nf
[...]
call EvolvePDF(dglap_h, initial_pdf, coupling, Q_init, Q_end &
                      & [, muR_Q] [, nloop] [, untie_nf] [, du] )
```

which takes a PDF array `pdf` and uses the splitting matrices in `dglap_h` to evolve it from scale `Q_init` to scale `Q_end`. By default the renormalisation to factorisation scale ratio is `muR_Q` = 1.0 and the number of loops in the evolution is the same as was used for the running `coupling` (the `nloop` optional argument makes it possible to override this choice). Variable flavour-number switching takes place at the quark masses as associated with the `coupling`. The choice to switch at the pole masses or $\overline{\rm MS}$ masses is the same as was specified by the user for the coupling.

If the `dglap_holder` object `dglap_h` does not support the relevant number of loops or flavours, the program will give an error message and stop. With the `untie_nf` option you can request that the number of flavours in the evolution be 'untied' from that in the coupling in the regions where `dglap_h` does not support the number of flavours used in the coupling. Instead the closest number of flavours will be used.[12]

[12]For example if `dglap_h` was initialised with $n_f = 3\ldots5$ while the coupling has $n_f = 3\ldots6$, then variable flavour number evolution will be used up to $n_f = 5$, but beyond the top mass the evolution will carry on with 5 flavours, while the coupling uses 6 flavours. There probably aren't too many good reasons for doing this (other than for examining how much it differs from a 'proper' procedure).

Mass thresholds (NNLO) are implemented as described in section 2:

$$\texttt{pdf}_{n_f} = \texttt{pdf}_{n_f-1} + \left(\frac{\alpha_s^{(n_f)}(x_\mu m_h^2)}{2\pi}\right)^2 (\texttt{dglap\_h\%MTM2 .conv. pdf}_{n_f-1}) \,, \tag{27a}$$

$$\texttt{pdf}_{n_f-1} = \texttt{pdf}_{n_f} - \left(\frac{\alpha_s^{(n_f)}(x_\mu m_h^2)}{2\pi}\right)^2 (\texttt{dglap\_h\%MTM2 .conv. pdf}_{n_f}) \,, \tag{27b}$$

when crossing the threshold upwards and downwards, respectively. Note that the two operations are not perfect inverses of each other, because the number of flavours of the `pdf` used in the convolution differs in the two cases. The mismatch however is only of order α_s^4 (NNNNLO), i.e. well beyond currently known accuracies.

A general remark is that crossing a flavour threshold downwards will result in some (almost certainly physically spurious) intrinsic heavy-flavour being left over below threshold.

6.2.2 Precomputed evolution and the `evln_operator`

Each Runge-Kutta evolution step involves multiple evaluations of the derivative of the PDFs, and the evolution between two scales may be broken up into multiple Runge-Kutta steps. This amounts to a large number of convolutions. It can therefore be useful to create a single *derived* splitting matrix that is equivalent to the whole evolution between the two scales.

A complication arises because evolutions often cross flavour thresholds, whereas a derived splitting matrix is only valid for fixed n_f. Therefore a new type has to be created, `evln_operator`, which consists of a linked list of splitting and mass threshold matrices, breaking an evolution into a chain of interleaved fixed-flavour evolution steps and flavour changing steps. An `evln_operator` is created with a call that is almost identical to that used to evolve a PDF:

```
type(evln_operator) :: evop
real(dp), pointer   :: pdf_init(:,:), pdf_end(:,:)
[...]
call InitEvlnOperator(dglap_h, evop, coupling, Q_init, Q_end &
                        & [, muR_Q] [, nloop] [, untie_nf] [, du] )
```

It can then be applied to PDF in the same way that a normal `split_mat` would:

```
pdf_end = evop * pdf_init       ! assume both pdfs already allocated
! OR (alternative form)
pdf_end = evop .conv. pdf_init
```

As usual the `Delete` subroutine can be used to clean up any memory associated with an evolution operator that is no longer needed.

7 Tabulated PDFs

The tools in the previous section are useful if one knows that one needs DGLAP evolution results at a small number of predetermined Q values. Often however one simply wishes to

provide a PDF distribution at some initial scale and then subsequently be able to access it at arbitrary values of x and Q. For this purpose it is useful (and most efficient) to produce a table of the PDF as a function of Q, which then allows for access to the PDF at arbitrary x and Q using an interpolation. All types and routines discussed in this section are in the `pdf_tabulate` module, or accessible also from `hoppet_v1`.

7.1 Preparing a PDF table

The type that contains a PDF table is `pdf_table`. It first needs to be allocated,

```
type(pdf_table) :: table
[...]

call AllocPdfTable(grid, table, Qmin, Qmax &
                   & [, dlnlnQ ] [, lnlnQ_order ] [, freeze_at_Qmin] )
```

where one specifies the range of Q values to be tabulated, from `Qmin` to `Qmax`, and optionally the interpolation step size `dlnlnQ` in the variable $\ln\ln Q/(0.1\text{ GeV})$ (default `dlnlnQ` $= 0.07$, sufficient for 10^{-3} accuracy), the interpolation order `lnlnQ_order`, equal to 3 by default, and finally whether PDFs are to be frozen below `Qmin`, or instead set to zero (the default is `freeze_at_Qmin=.false.`, i.e. they are set to zero).[13]

By default a tabulation knows nothing about n_f thresholds, which means that in the neighbourhood of thresholds the tabulation would be attempting to interpolate a discontinuous function (at NNLO). To attribute information about n_f thresholds to the tabulation, set them up first in a running `coupling` object and then transfer them:

```
call AddNfInfoToPdfTable(table,coupling)
```

When interpolating the table (see below), the set of Q values for the interpolation will be chosen so as to always have a common n_f value. Note that `AddNfInfoToPdfTable` may only be called once for an allocated table: if you need to change the information about n_f thresholds, `Delete` the table, reallocate it and then reset the n_f information. This is not necessary if you just change the value of the coupling.

Given an existing table, `ref_table`, a new table, `new_table`, can be allocated with identical properties (including any n_f information) as follows

```
call AllocPdfTable(new_table, ref_table)
```

All of the above routines can be used with 1-dimensional arrays of tables as well (in `AllocPdfTable` the reference table must always be a scalar).

A table can be filled either from a routine that provides the PDFs as a function of x and Q, or alternatively by evolving a PDF at an initial scale. The former can be achieved with

[13] Note that the spacing and interpolation in Q are treated independently of what's done in the PDF evolution. A reason for this is that in the PDF evolution one uses a variable related to the running coupling, which is similar to $\ln\ln Q$ but whose precise details depend on the particular value of the coupling. Using this in the tabulation would have prevented one from having a tabulation disconnected from any coupling. Unfortunately `du` and `dlnlnQ` are not normalised equivalently — roughly speaking for $n_f = 4$ they correspond to the same spacing if `dlnlnQ` $\simeq 0.7$`du`.

```
call FillPdfTable_LHAPDF(table, LHAsub)
```

where `LHAsub` is any subroutine with the LHAPDF interface, *i.e.*, as shown earlier in section 5.1.1.

To fill a table via an evolution from an initial scale, one uses

```
type(pdf_table)        :: table
type(dglap_holder)     :: dglap_h
type(running_coupling) :: coupling
real(dp), pointer      :: initial_pdf(:,:)
real(dp)               :: Q0
integer                :: nloop
integer                :: untie_nf
[...]
call EvolvePdfTable(table, Q0, initial_pdf, dglap_h, coupling &
                    & [, muR_Q] [, nloop] [, untie_nf] )
```

which takes an the `initial_pdf` at scale `Q0`, and evolves it across the whole range of Q values in the table, using the `EvolvePDF` routine. The arguments have the same meaning as corresponding ones in `EvolvePDF`, explained in section 6.2.1. The `du` value that's used is the default one for `EvolvePDF` which, we recall, may be set using `SetDefaultEvolutionDu(du)`. If the Q spacing in the tabulation is such that steps in `du` would be too large, then the steps are automatically resized to the tabulation spacing.

One may also use the precomputed evolution facilities of section 6.2.2, by calling the routine

```
call PreEvolvePdfTable(table, Q0, dglap_h, coupling, &
                       & [, muR_Q] [, nloop] [, untie_nf] )
```

which prepares `evln_operator`s for all successive Q intervals in the table. An accelerated evolution, which uses these operators instead of explicit Runge-Kutta steps, may then be obtained by calling

```
call EvolvePdfTable(table, initial_pdf)
```

The `EvolvePdfTable` routine may be called as many times as one likes, for different initial PDFs for example; however, if one wishes to change the parameters of the evolution (coupling, perturbative order, etc.) in the precomputed option, one must first `Delete` the table and then prepare again it.

7.2 Accessing a table

The main way to access a table is as follows

```
real(dp) :: pdf(-6:6), y, x, Q
  [...]
  call EvalPdfTable_yQ(table, y, Q, pdf)
  ! or using x
  call EvalPdfTable_xQ(table, x, Q, pdf)
```

There may be situations where it is useful to access the internals of a `pdf_table`, for example because one would like to carry out a convolution systematically on the whole contents of the table. Among the main elements are

```
type pdf_table
  integer           :: nQ           ! arrays run from 0:nQ
  real(dp), pointer :: tab(:,:,:)   ! the actual tabulation
  real(dp), pointer :: Q_vals(:)    ! the Q values
  integer,  pointer :: nf_int(:)    ! nf values at each Q
  real(dp), pointer :: as2pi(:)     ! alphas(Q)/2pi at each Q
  [...]
end type pdf_table
```

where the third dimension of `tab` spans the range of tabulated Q values, and the n_f and coupling information are only allocated and set if one has called `AddNfInfoToPdfTable` for the table.

An example of usage of the low-level information contained in the table is the following, which initialises `table_deriv_LO` with the LO derivative of `table`:

```
do iQ = 0, table%nQ
  table_deriv_LO%tab(:,:,iQ) = table%as2pi(iQ) * &
                  & ( dglap_h%allP(1,table%nf_int(iQ)) * table%tab(:,:,iQ))
end do
```

where we assume `table_deriv_LO` to have been allocated with an appropriate structure at some point, e.g. via

```
call AllocPdfTable(table_deriv_LO, table)
```

The above mechanism has found use in the a-posteriori PDF library [17, 18] and in work matching event shapes with fixed-order calculations [15, 14]. One could also imagine using it to obtain tables of (flavour-separated) structure functions, if one were to convolute with coefficient functions rather than splitting functions.

There are instances in which it is useful to extract the PDF at all x values for a given Q. This is the case, for efficiency reasons, if one wishes to evaluate it at very many x values and a single Q value. It's also the case if one wishes to evaluate sum rules. One then uses the routine `EvalPdfTable_Q`, as in the following example for calculating the momentum sum rule

```
real(dp), pointer :: pdf_at_Q(:,:)
real(dp)          :: moment_index, momentum_sum
! [...]
call AllocPDF(table%grid, pdf_at_Q)        ! reserve space
call EvalPdfTable_Q(table, Q, pdf_at_Q)    ! evaluate the PDF at Q
moment_index = 1.0_dp
! (truncated) momentum-sum of the sum of all flavours of pdf_at_Q
momentum_sum = TruncatedMoment(table%grid, moment_index, &
                            & sum(pdf_at_Q(:,-6:6),dim=2)) ! dim=2 -> sum over flav
```

where the `TruncatedMoment` function was described in detail in section 4.4. Recall that if the goal is to extract accurate sum-rule estimates or in general full (non-truncated) moments, the value of `ymax` of the grid should be large enough.

As with all other objects, a `pdf_table` object can be deleted using

```
call Delete(table)
```

Notice that a table is a variable that is local to the scope in which it is defined. So table variables defined separately in each of two different procedures will effectively be different variables in the two procedures. If ones needs to use a common PDF table across different procedures, it has to be defined within a module and then the module should be `use`d in both procedures. In Appendix A, there is a detailed example of different ways of accessing a table.

8 Streamlined interface

Now we present the streamlined interface to HOPPET, intended to allow easy access to the essential evolution functionality from languages other than F95. It hides all the object-oriented nature of the program, and provides access to one `pdf_table`, based on a single grid definition. The description will be given as if one is calling from F77. An include file `src/hoppet_v1.h` is provided for calling from `C++` — the interface is essentially identical to the Fortran one, with the caveat that names are case sensitive (the cases are as given below), and that PDF components referred to below as `pdf(-6:6)` become `pdf[0..12]`. A summary of the most relevant procedures of this interface and their description can be found in the reference guide, Appendix B.

8.1 Initialisation

The simplest way of initialising the streamlined interface is by calling

```
call hoppetStart(dy,nloop)
```

which will set up a compound (four different spacings) grid with spacing `dy` at small x, extending to $y = 12$, and numerical order `=` -6. The Q range for the tabulation will be 1 GeV $< Q <$ 28 TeV and a reasonable choice will be made for the `dlnlnQ` spacing (related to `dy`). It will initialise splitting functions up to `nloop` loops (though one can still carry out evolutions with fewer loops). If you need more control over the initialisation, you should use

```
call hoppetStartExtended(ymax,dy,Qmin,Qmax,dlnlnQ,nloop,order,factscheme)
```

which will again set up compound grid, but give control over the numerical `order` and the y and Q ranges and spacings (as before `dy` is the spacing at small x). It also allows one to choose the type of evolution according to `factscheme`.

8.2 Usage

To carry out an evolution, one should first decide whether one wants a fixed-flavour number scheme or a variable flavour number scheme (the default). Either can be set with its parameters as follows:

```
call hoppetSetFFN(fixed_nf)
call hoppetSetPoleMassVFN(mc, mb, mt)  ! Heavy quark pole masses
call hoppetSetMSbarMassVFN(mc, mb, mt) ! Heavy quark MSbar masses, m(m)
```

where for the VFNs one specifies either the pole-masses or the $\overline{\rm MS}$ masses for the quarks (thresholds are then implemented at the pole or $\overline{\rm MS}$ masses accordingly). An evolution is carried out with the following routine

```
call hoppetEvolve(asQ, Q0alphas, nloop, muR_Q, LHAsub, Q0pdf)
```

where one specifies the coupling `asQ` at a scale `Q0alphas`, the number of loops for the evolution, `nloop`, the ratio of renormalisation to factorisation scales `muR_Q`,[14] the name of a subroutine `LHAsub` with interface

```
subroutine LHAsub(x,Q,pdf)
  implicit none
  double precision x,Q,pdf(-6:6)
  [...]   ! sets pdf to be momentum densities, e.g. pdf(0) = xg(x)
end subroutine
```

to return the initial condition for the evolution and the scale `Q0pdf` at which one starts the PDF evolution. Note that the LHAsub subroutine will only be called with `Q` = `Q0pdf`. To access the coupling one uses

```
alphas = hoppetAlphaS(Q)
```

while the PDF at a given value of `x` and `Q` is obtained with

```
call hoppetEval(x,Q,f)
```

which sets `f(-6:6)` (recall that it is $xg(x)$, etc., that is returned, since this is what is used through HOPPET).

It is also possible to prepare an evolution in cached form. This is useful if one needs to evolve many different PDF sets with the same evolution properties (coupling, initial scale, etc.), as is the usual situation in global analyses of PDFs, because though the preparation may take a bit longer than a normal evolution (2–10 times depending on the `order`), once it is done, cached evolutions run 3–4 faster than a normal evolution. The preparation of the cache is carried out with

```
call hoppetPreEvolve(asQ, Q0alphas, nloop, muR_Q, Q0pdf)
```

and then the cached evolution is carried out with

```
call hoppetCachedEvolve(LHAsub)
```

[14] Note that in the streamlined interface, with `muR_Q` $\neq 1$, the running coupling flavour thresholds are still placed at the quark masses; the evolution needs the coupling for a given number of flavours outside the standard range for that number of flavours (precisely because `muR_Q` $\neq 1$) and this is done automatically in the evolution. In contrast in the benchmark studies [19], the flavour thresholds for the coupling were placed at `muR_Q` $\times m_Q$. This is a perfectly valid alternative, but can complicate the specification of the α_s value — for example with `muR_Q` $= 0.5$ the matching for the top threshold would be carried out at $\mu = 0.5 m_t \simeq 85$ GeV, and if one specified the coupling at scale M_Z, it wouldn't be clearer whether this was a 5-flavour value or a 6-flavour value. With the procedure adopted in the streamlined interface the issue does not arise. (While in F95 the user has the freedom to do as they prefer).

The results may be very slightly different from those in a normal evolution (some information is lost when caching), and the user may wish to check on a case-by-case basis that such differences don't matter in practice.

The tabulation can also be filled with the contents of an external PDF package, by calling

```
call hoppetAssign(LHAsub)
```

where `LHAsub` is the name of any subroutine with the interface given above, which will now be called with a range of `Q` values corresponding to the internal tabulation scales. This essentially just transfers an external tabulation into HOPPET's internal representation.

Finally given an evolved or assigned PDF, one can obtain information about convolutions of the splitting functions with the PDFs:

```
call hoppetEvalSplit(x,Q,iloop,nf,f)
```

sets `f(-6:6)` equal to the value at `x`, `Q` of convolution of the `iloop` splitting function matrix (with `nf` flavours) with the currently tabulated PDF. If `nf` < 0 the number of flavours used is the one appropriate at the specified Q scale (as long as the information is available, i.e. one of `hoppetEvolve` or `hoppetCachedEvolve` has been called). The first call with a given `nf` for a specified `iloop` will be slow ($\sim$ the time for a cached evolution), but subsequent calls with the same values will be fast.

The routines described here are to be found in `src/streamlined_interface.f90` and may provide inspiration for the user wishing to write their own F95 code for HOPPET.

9 Benchmarks

Key questions in assessing the usefulness of a PDF evolution code include that of its correctness, its accuracy and its speed. HOPPET's correctness has been established with a reasonable degree of confidence in the benchmark tests [19] where it was compared with the Mellin space based evolution code QCD-Pegasus [3]. The program used to carry out those tests is available as `benchmarks/benchmarks.f90`. The user should carefully read the detailed comments at the beginning for usage instructions.

The results used in [19] were obtained with very finely spaced grids, in order to guarantee small numerical errors ($\lesssim 10^{-7}$). Such accuracies are useful when comparing and testing two independent codes, because differences or bugs in the implementation of the physics (especially the higher-order parts) may only manifest themselves as small changes in the results.

In contrast, for use in most physical applications, an accuracy in the range 10^{-3} to 10^{-4} is generally more than adequate, since it is rare for other sources of numerical uncertainty (e.g. Monte Carlo integration errors in NLO codes, or experimental errors) to be comparably small. The critical issue in such cases is more likely to be the speed of the code, for example in PDF fitting applications.

HOPPET's accuracy and speed both depend on the choice of grid (in y) and the evolution and/or tabulation steps in Q. We shall start with the question of the accuracy.

9.1 Accuracy

To measure the accuracy, we use the same initial condition and evolution parameters as in [19]:

$$xu_v(x) = 5.107200x^{0.8}(1-x)^3\,, \tag{28a}$$
$$xd_v(x) = 3.064320x^{0.8}(1-x)^4\,, \tag{28b}$$
$$x\bar{d}(x) = 0.1939875x^{-0.1}(1-x)^6\,, \tag{28c}$$
$$x\bar{u}(x) = x\bar{d}(x)(1-x)\,, \tag{28d}$$
$$xs(x) = x\bar{s}(x) = 0.2(x\bar{d}(x) + x\bar{u}(x))\,, \tag{28e}$$
$$xg(x) = 1.7x^{-0.1}(1-x)^5\,, \tag{28f}$$

where $u_v \equiv u - \bar{u}$, $d_v \equiv d - \bar{d}$, and all other flavours are zero. The initial scale[15] is $Q_0 = \left(\sqrt{2} - \bar{\epsilon}\right)$ GeV, $\alpha_s(Q_0) = 0.35$ and the charm, bottom and top pole masses are kept at the default values, $\sqrt{2}$, 4.5 and 175 GeV respectively (as used also in the streamlined interface). Observe that the initial conditions and coupling are actually both given for three active flavours (i.e. infinitesimally below the charm mass). The evolution is carried out to NNLO accuracy in a variable flavour number scheme, including the mass thresholds in the coupling and PDF.

All tests here are carried out based on tabulations of the PDF evolution, section 7. We first run HOPPET with a very fine grid spacing to provide a reference result. Then we run the evolution for a coarser grid — the accuracy of the coarser grid is determined by comparing its results with those from the reference grid. We determine the relative accuracy for each flavour at 5000 points in the x, Q plane, as shown in Fig. 2. The points are uniformly spaced in $\zeta = \ln 1/x + 9(1-x)$ so as to obtain fine coverage at small and large x. The Q values are chosen more closely spaced at low Q where the Q-dependence is strongest and they are taken slightly correlated with ζ so as to cover a nearly continuous range of Q.[16]

One difficulty that arises when examining relative accuracies is that some flavours change sign as one varies x or Q. Close to the zero the relative accuracy diverges because of the small value of the denominator. Therefore in global accuracy estimates, we eliminate flavours in the region where they change sign (within $\Delta\zeta = 0.4$ and at the two Q values straddling a sign change). Specifically, for our initial conditions, this corresponds to $c, \bar{c}$ for the two lowest Q values below $x \sim 10^{-2}$ and $\bar{u}$ for $x \gtrsim 0.9$.[17] The exact regions are shaded in grey in Fig. 2.

[15] With $\bar{\epsilon}$ an infinitesimal number. Note that this is unrelated to the evolution accuracy ϵ, introduced later in this section.

[16] This procedure differs from that in [19] where fewer (500) points were used and one compared not individual flavours, but combinations intended to be more directly revealing of any deficiencies in the evolution. This reflects the difference in needs between obtaining a global measure of the accuracy and providing benchmarks intended in part to facilitate the debugging of independent codes.

[17] The change in sign of the charm distribution is not worrying physically since it is close to threshold where it will be compensated for by finite mass effects in the coefficient functions; for $\bar{u}$ the sign change is more surprising, though it may be related to non-trivial interactions between the evolutions of the u

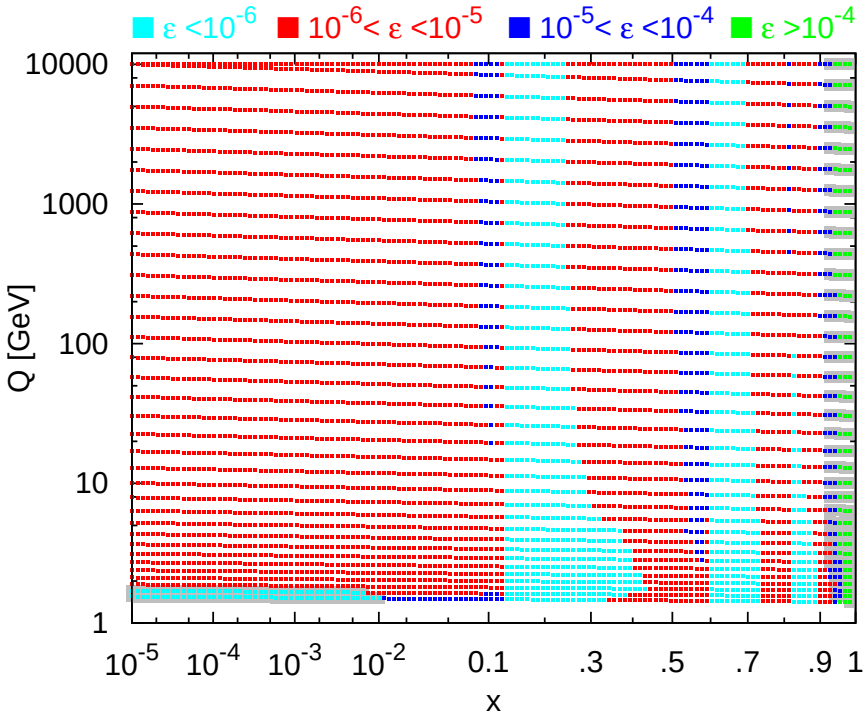

Figure 2: The set of points in x, Q used to determine the accuracy of the evolution. The areas shaded in grey are regions where one of the flavours is in the neighbourhood of a sign-change (bottom-left: $c, \bar{c}$, right: $\bar{u}$) and so is ignored in the accuracy determination. The accuracies shown here correspond to a y grid with a base spacing of `dy` $= 0.2$ and other parameters as described in the text for Fig. 3. The colour coding indicates the error in the least-well determined (non-excluded) flavour channel ($\bar{b} \ldots b$) at each point.

Our tabulation covers the range $10^{-5} < x < 1$, $\sqrt{2} < Q < 10^4$ GeV. The grid in $y = \ln 1/x$ will consist of 4 nested subgrids: one covering the whole y range with spacing `dy` and others with spacings `dy`$/3, $`dy`$/9, $`dy`$/27$ extending to $y = 2, 0.5, 0.2$ respectively. Except where stated we shall use `order` $= -6$. In Q the default interpolation order will be 4. The reference grids use `dy` $= 0.025$ and `dlnlnQ` $= 0.005$.

Fig. 3 shows the relative accuracy ϵ as a function of x for two grid-spacing choices (left `dy` $= 0.2$, right `dy` $= 0.05$, `dlnlnQ` $=$ `dy`$/4$ in both cases). Each solid line corresponds to one Q value and shows the error in the least-well-determined flavour channel at each x, excluding flavour channels close to a sign-change. The relative accuracy ϵ is poorest as one approaches $x = 1$, where the PDFs all go to zero very rapidly and so have divergent logarithmic derivatives in x, $d \ln q / d \ln x$, adversely affecting the accuracy of the convolutions. This region is always the most difficult in x–space methods, however the use of multiple subgrids in x allows to one to obtain acceptable results for $x < 0.9$ which is likely to be the largest value of any phenomenological relevance.

At $x \sim 0.1$, 0.6 and 0.8 one notices step-like structures — these are the points where one switches between subgrids, with a significant degradation in accuracy at x values below the transition. These structures are also visible in the colour-coded accuracy representation in Fig. 2, which corresponds to `dy` $= 0.2$ and allows one to visualise more clearly the Q dependence of the accuracy. The effect of the grid spacing is clearly visible as one goes from the left to the right-hand plots of Fig. 3, with the reduction in the spacings by a factor of 4 leading to an improvement in accuracy by a factor ~ 100.

For completeness we also show the parts of the charm channel that have been excluded because of the proximity to a sign change (dashed lines, lower-left shaded region of Fig.2). One observes in particular a spike near $x \simeq 7 \times 10^{-3}$ where the charm distribution has its zero. Including this in a estimate of the global accuracy would be senseless since it actually corresponds to a divergence and the peak-value for the spike is arbitrary, depending on the precise choice of points used to estimate the accuracy. The question of the exact region to exclude is somewhat arbitrary, but the choice made above seems not unreasonable in the light of Fig. 3.

Fig. 3 is useful in order to obtain a detailed picture of the accuracy of the evolution with a given set of parameters. To quote a single, global, number for the relative accuracy ϵ we make the conservative choice of taking the largest value of ϵ that occurs in a chosen x range. We will examine a restricted range, $x < 0.7$, studying just the g, u, d, s flavours, and also a wider range, $x < 0.9$ with all flavours.

Fig. 4 shows the effect of varying the base `dy` and `dlnlnQ` separately, while the other is fixed at the reference value. The 'guds' flavours in the $x < 0.7$ range are generally better determined, for a given set of grid parameters, than the full set of flavours up to $x < 0.9$. This is as one would expect since the large-x region is usually the hardest and the 'guds' flavours are generally somewhat smoother than the others. Using only three y subgrids worsens the situation when including the largest x values, and reducing the order in the Q

and $\bar{u}$ components at NNLO (note that in the region of the sign change they differ by many orders of magnitude).

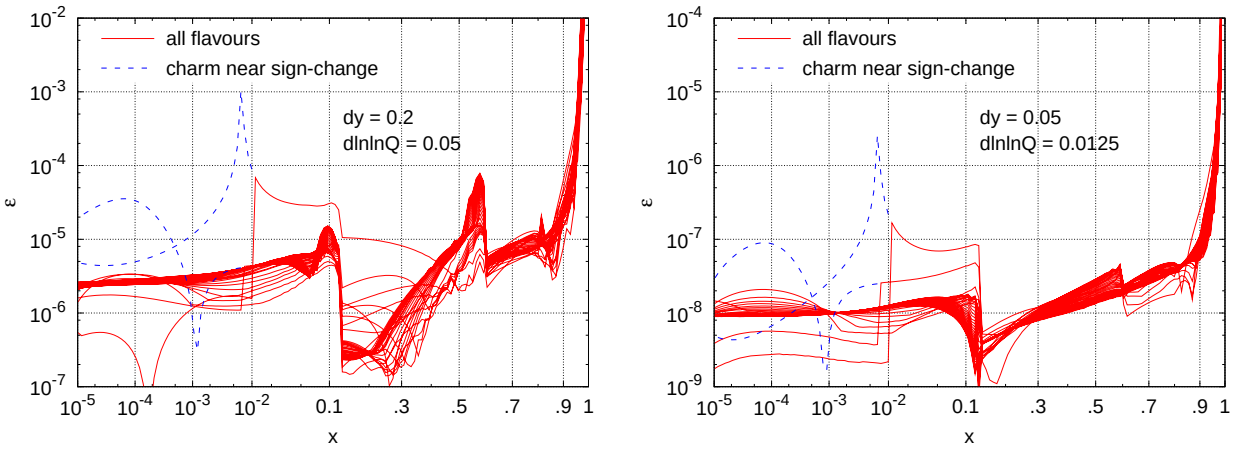

Figure 3: The relative accuracy ϵ of the least well determined flavour channel at each x, Q point, shown as a function of x for many Q values. The results for the part of the charm distribution excluded from the analysis (near sign change) are shown separately.

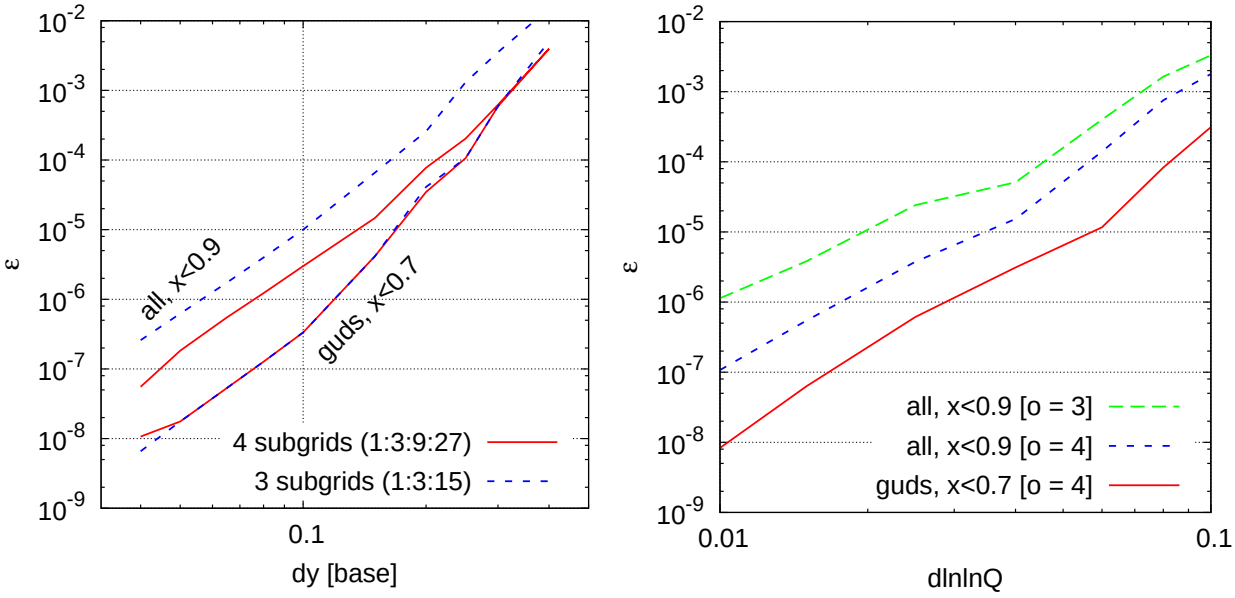

Figure 4: Left: the (globally) worst relative accuracy ϵ as a function of the base y-grid resolution parameter, `dy` — shown for two y-grid configurations and two x-ranges and flavour-sets. Right: the relative accuracy as a function of the resolution in $\ln\ln Q$ of the tabulation, `dlnlnQ`, for different x/flavour ranges and for different `lnlnQ_order` values (`o`).

		lf95	ifort	g95
t_s	[s]	0.9	0.66	2.8
t_α	[ms]	0.16	0.12	0.13
t_i	[ms]	37	38	330
t_p	[ms]	51	44	310
t_c	[ms]	8.8	9.8	110
t_{xQ}	[μs]	2.7	3.1	25

Table 2: Contributions to the run time in eqs. (29) for `dy` = 0.2 and `dlnlnQ` = 0.05 and standard values for the other parameters (on a 3.4GHz Pentium IV (D) with 2 MB cache).

interpolation also adversely affects the accuracy (also for $x < 0.7$, not shown).

From Fig. 4, we deduce that inaccuracies from the Q and y parts of the grid are similar when `dlnlnQ` = `dy`/4. This is the combination that we shall use as standard.

9.2 Timing

The time spent in HOPPET for a given analysis can expressed as follows, according to whether or not one carries out pre-evolution:

$$t_{\text{no pre-ev}} = t_s + n_\alpha t_\alpha + n_i(t_i + n_{xQ}\, t_{xQ})\,, \tag{29a}$$

$$t_{\text{with pre-ev}} = t_s + n_\alpha(t_\alpha + t_p) + n_i(t_c + n_{xQ}\, t_{xQ})\,, \tag{29b}$$

where t_s is the time for setting up the splitting functions, n_α is the number of different running couplings that one has, t_α is the time for initialising the coupling, n_i is the number of PDF initial conditions that one wishes to consider, t_i is the time to carry out the tabulation for a single initial condition, n_{xQ} is the number of points in x, Q at which one evaluates the full set of flavours once per PDF initial condition; in the case with pre-prepared cached evolution, t_p is the time for a preparing a cached evolution and t_c is the time for performing the cached evolution. Finally t_{xQ} is the time it takes to evaluate the PDFs at a given value of (x, Q) once the tabulation has been performed.

The various contributions to the run-time are shown in table 2 for `dy` = 0.2 and `dlnlnQ` = 0.05 (giving an accuracy $\sim 10^{-4}$), for various compilers. In a typical analysis where run-times matter, such as a PDF fit, it is to be expected that the time will be dominated by t_c (or t_i), or if the number of x, Q points is rather large ($\gtrsim 3000$), by $n_{xQ} t_{xQ}$.[18] We note (with regret) the considerable speed advantage (almost an order of magnitude) that is to be had with commercial compilers.[19]

We study t_c and t_i in more detail in Fig. 5, where we relate them to the accuracy obtained from the evolution. As one would expect, studying just the 'guds' flavours for

[18] With these numbers, it is easy to check that a global fit with $n_{xQ} \sim 3000$ and $n_i \sim 10^5$ could be completed in less than half an hour, assuming cached evolution and the use of commercial compilers.

[19] As this article was going to press, a development version of gfortran (4.4) became available that gives correct results with HOPPET and is found to be comparable in speed to commercial compilers.

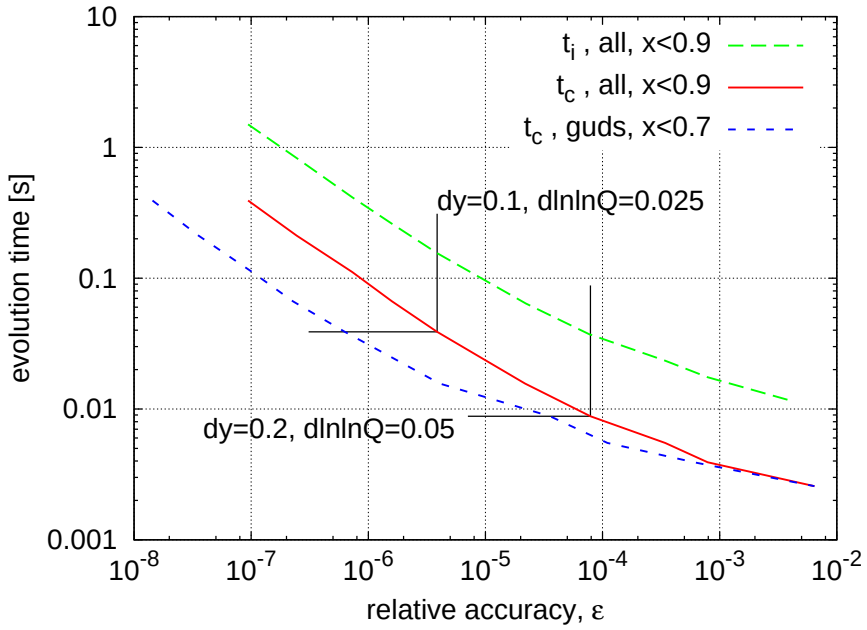

Figure 5: relative accuracy obtained as a function of the time taken to perform the evolution (lf95, 3.4GHz Pentium IV (D) with 2 MB cache).

$x < 0.7$ one obtains better accuracy for a given speed than with all flavours for $x < 0.9$. Overall one can obtain 10^{-4} accuracy with $t_c \simeq 10^{-2}$ s and 10^{-6} accuracy with $t_c \simeq 10^{-1}$ s. In general t_i is about $4-5$ larger than t_c, highlighting the advantage of the cached evolution.

We note that the time t_{xQ} for evaluating each point is essentially independent of `dy` and `dlnlnQ`. If a computation is dominated by t_{xQ}, then it can be made somewhat faster by lowering the interpolation orders, at the expense of needing a finer grid (and so longer evolution times).

The timings shown here are roughly similar, for accuracies $\sim 10^{-4}$, to those obtained with the N–space code Pegasus [3] when the number of x, Q points to be evaluated is $\mathcal{O}(10^3)$. For much smaller numbers of points Pegasus becomes superior (because of the significantly smaller ratio t_c/t_{xQ}), while for much larger numbers of points HOPPET becomes better. Other NNLO evolution codes published in recent years [5, 6, 1] are generally less competitive either in terms of accuracy or speed.

To close this section, we summarise in table 3 the different parameters that are relevant in determining the accuracy of the evolution and tabulation, together with comments about the components of the timing affected by each parameter.

10 Conclusions

HOPPET is an x–space evolution code that is novel both in terms of the accuracy and speed that it provides compared to other x–space codes, and in terms of its interface, designed to provide a straightforward and physical way of manipulating PDFs beyond the built-in

Parameter	Default	Timing impact	Notes
base `dy`	–	t_s, t_i, t_p, t_c	Default subgrids in ratio 1:3:9:27
`order`	$[-6]$	$t_s, t_p, t_{xQ}(t_i, t_c)$	
`DefaultConvolutionEps`	10^{-7}	t_s	final acc. limited by $\sim$ twice this.
`du`	0.1	t_i, t_p	immaterial if $\gtrsim 1.4\,$`dlnlnQ`
`dlnlnQ`	[`dy`/4]	t_i, t_p, t_c	
`lnlnQ_order`	4	t_{xQ}	
`DefaultCouplingDt`	0.2	t_α	default sufficient for $\epsilon \sim 10^{-9}$

Table 3: Parameters involved in the accuracy of a tabulated evolution. Default values shown in square brackets are to be specified by hand in the F95 interface, but are automatically set in the simpler of the initialisation calls with the streamlined interface.

task of DGLAP evolution.

Features that might be envisaged for future releases include DIS coefficient functions, full support for the DIS factorisation scheme, and the addition of time-like evolution, relevant for phenomenological fits to fragmentation functions as in [41]. In principle, the information presented here is sufficient to allow a user to implement the coefficient functions themselves, while the DIS scheme and timelike evolution would require somewhat more knowledge of the internals of the program.

More ambitious possible extensions cover a wide range of physics. Just within QCD, a general physical feature absent from mainstream PDF evolution codes is that of evolution that includes matching with various types of resummed calculations. Although studies in this direction have already been performed, both for small x resummations, as in [39] and for large x resummation, as in [42], no general public code exists which performs a matching between resummed and fixed (NLO, NNLO) order splitting functions, either in the time-like case or in the space-like case. In particular, work to extend HOPPET in order to deal with general interpolated splitting functions and coefficient functions, required to implement small-x resummation, is in progress.

Another interesting extension of HOPPET would be to implement a more general mass treatment of heavy quarks. A proper treatment of heavy quark mass effects is required to obtain a good description of heavy flavour structure function as measured in HERA. Although there are by now several studies of the effect of heavy quark masses in global analysis of PDFs [43, 44], which show sizable effects on predictions for LHC observables, there does not exist right now a public code were this General Mass heavy quark schemes are implemented.

Also of interest are non-QCD effects. Evolutions with QED radiation have been presented in [5, 45], however so far no public code exists for evolution including both QCD and electroweak (EW) effects [46, 47]. This is of particular relevance at LHC energies, since flavour is associated with an SU(2) charge and so soft divergences (above M_W) do not cancel in the PDF evolution between real and virtual contributions, leading one to expect non-negligible effects in the flavour structure of the PDFs at high scales. The full

QCD+EW evolution is a rather task complicated because of the need to include the EW flavour couplings, including the CKM matrix, and polarisation. For this kind of problem a code such as HOPPET provides a good starting point, since it has a clean separation of the numerical and flavour aspects of evolution, and verified unpolarised and polarised evolution.

Acknowledgements

This work was initiated in the context of DIS event shape resummation and matching studies [14] with Mrinal Dasgupta. Its subsequent development into a fully-featured NNLO evolution code owes much to Andreas Vogt's regular encouragement, and his suggestions about features that would be useful to include for benchmark tests. We are grateful also to Wu-Ki Tung for comments on the documentation. This work was supported in part by grant ANR-05-JCJC-0046-01 from the French Agence Nationale de la Recherche.

A Example programs

A.1 General interface

The program below generates a subset of table 15 of the NNLO benchmark evolution in the second reference of [19]. It is to be found (in a slightly more commented form) in `example_f90/tabulation_example.f90`. Compilation instructions are to be found in the `README` file in the main directory of the release. A program that has the same functionality but in F77, using the streamlined interface of section 8, is to be found as `example_f77/tabulation_example.f` (cf. section A.2).

```
program tabulation_example
  use hoppet_v1
  implicit none
  real(dp)         :: dy, ymax, quark_masses(4:6)
  integer          :: order, nloop, ix
  type(grid_def)   :: grid, gdarray(4) ! holds information about the grid
  type(dglap_holder)     :: dh         ! holds the splitting functions
  type(pdf_table)        :: table      ! holds the PDF tabulation
  type(running_coupling) :: coupling
  real(dp), pointer      :: pdf0(:,:)  ! holds the initial pdf
  real(dp)               :: Q0, Q, pdf_at_xQ(-6:6)
  real(dp), parameter :: heralhc_xvals(9) = &
       & (/1e-5_dp,1e-4_dp,1e-3_dp,1e-2_dp,0.1_dp,0.3_dp,0.5_dp,0.7_dp,0.9_dp/)

  ! set up parameters for grid
  order = -6
  ymax  = 12.0_dp
  dy    = 0.1_dp

  ! set up the grid itself -- we use 4 nested subgrids
```

```
call InitGridDef(gdarray(4),dy/27.0_dp, 0.2_dp, order=order)
call InitGridDef(gdarray(3),dy/9.0_dp,  0.5_dp, order=order)
call InitGridDef(gdarray(2),dy/3.0_dp,  2.0_dp, order=order)
call InitGridDef(gdarray(1),dy,          ymax , order=order)
call InitGridDef(grid,gdarray(1:4),locked=.true.)

! initialise the splitting-function holder
nloop = 3
call InitDglapHolder(grid,dh,factscheme=factscheme_MSbar,&
     &                     nloop=nloop,nflo=3,nfhi=6)

! initialise a PDF from the function below (must be contained,
! in a "used" module, or with an explicitly defined interface)
call AllocPDF(grid, pdf0)
pdf0 = unpolarized_dummy_pdf(xValues(grid))
Q0 = sqrt(2.0_dp)  ! the initial scale

! allocate and initialise the running coupling with a given
! set of quark masses (NB: charm mass just above Q0).
quark_masses(4:6) = (/1.414213563_dp, 4.5_dp, 175.0_dp/)
call InitRunningCoupling(coupling,alfas=0.35_dp,Q=Q0,nloop=nloop,&
     &                   quark_masses = quark_masses)

! create the tables that will contain our copy of the user's pdf
! as well as the convolutions with the pdf.
call AllocPdfTable(grid, table, Qmin=1.0_dp, Qmax=10000.0_dp, &
     & dlnlnQ = dy/4.0_dp, freeze_at_Qmin=.true.)
! add information about the nf transitions to the table (improves
! interpolation quality)
call AddNfInfoToPdfTable(table,coupling)

! create the tabulation based on the evolution of pdf0 from scale Q0
call EvolvePdfTable(table, Q0, pdf0, dh, coupling, nloop=nloop)
! alternatively "pre-evolve" so that subsequent evolutions are faster
!call PreEvolvePdfTable(table, Q0, dh, coupling)
!call EvolvePdfTable(table,pdf0)

! get the value of the tabulation at some point
Q = 100.0_dp
write(6,'(a,f8.3,a)') "           Evaluating PDFs at Q = ",Q," GeV"
write(6,'(a5,2a12,a14,a10,a12)') "x",&
     & "u-ubar","d-dbar","2(ubr+dbr)","c+cbar","gluon"
do ix = 1, size(heralhc_xvals)
   call EvalPdfTable_xQ(table,heralhc_xvals(ix),Q,pdf_at_xQ)
   write(6,'(es7.1,5es12.4)') heralhc_xvals(ix), &
        &  pdf_at_xQ(2)-pdf_at_xQ(-2),  pdf_at_xQ(1)-pdf_at_xQ(-1), &
        &  2*(pdf_at_xQ(-1)+pdf_at_xQ(-2)), (pdf_at_xQ(-4)+pdf_at_xQ(4)), &
        &  pdf_at_xQ(0)
end do

! some cleaning up (not strictly speaking needed, but illustrative)
```

```
  call Delete(table); call Delete(pdf0); call Delete(dh)
  call Delete(coupling);  call Delete(grid)

contains
  !======================================================================
  !! The dummy PDF suggested by Vogt as the initial condition for the
  !! unpolarized evolution (as used in hep-ph/0511119).
  function unpolarized_dummy_pdf(xvals) result(pdf)
    real(dp), intent(in) :: xvals(:)
    real(dp)             :: pdf(size(xvals),-6:7) ! note upper bound!
    real(dp) :: uv(size(xvals)), dv(size(xvals))
    real(dp) :: ubar(size(xvals)), dbar(size(xvals))
    !---------------------
    real(dp), parameter :: N_g = 1.7_dp, N_ls = 0.387975_dp
    real(dp), parameter :: N_uv=5.107200_dp, N_dv = 3.064320_dp
    real(dp), parameter :: N_db = half*N_ls

    pdf = zero

    !-- remember that these are all xvals*q(xvals)
    uv = N_uv * xvals**0.8_dp * (1-xvals)**3
    dv = N_dv * xvals**0.8_dp * (1-xvals)**4
    dbar = N_db * xvals**(-0.1_dp) * (1-xvals)**6
    ubar = dbar * (1-xvals)

    ! labels iflv_g, etc., come from the hoppet_v1 module
    pdf(:, iflv_g) = N_g * xvals**(-0.1_dp) * (1-xvals)**5
    pdf(:,-iflv_s) = 0.2_dp*(dbar + ubar)
    pdf(:, iflv_s) = pdf(:,-iflv_s)
    pdf(:, iflv_u) = uv + ubar
    pdf(:,-iflv_u) = ubar
    pdf(:, iflv_d) = dv + dbar
    pdf(:,-iflv_d) = dbar
  end function unpolarized_dummy_pdf

end program tabulation_example
```

The expected output from the program is:

```
           Evaluating PDFs at Q =  100.000 GeV
    x      u-ubar      d-dbar    2(ubr+dbr)    c+cbar       gluon
1.0E-05  3.1907E-03  1.9532E-03  3.4732E+01  1.5875E+01  2.2012E+02
1.0E-04  1.4023E-02  8.2749E-03  1.5617E+01  6.7244E+00  8.8804E+01
1.0E-03  6.0019E-02  3.4519E-02  6.4173E+00  2.4494E+00  3.0404E+01
1.0E-02  2.3244E-01  1.3000E-01  2.2778E+00  6.6746E-01  7.7912E+00
1.0E-01  5.4993E-01  2.7035E-01  3.8526E-01  6.4466E-02  8.5266E-01
3.0E-01  3.4622E-01  1.2833E-01  3.4600E-02  4.0134E-03  7.8898E-02
5.0E-01  1.1868E-01  3.0811E-02  2.3198E-03  2.3752E-04  7.6398E-03
7.0E-01  1.9486E-02  2.9901E-03  5.2352E-05  5.6038E-06  3.7080E-04
9.0E-01  3.3522E-04  1.6933E-05  2.5735E-08  4.3368E-09  1.1721E-06
```

The file `example_f90/tabulation_example.default_output` contains a copy of these results, so as to allow easy comparison. The numbers correspond to evolution with variable

flavour number, $\mu_F = \mu_R$, and the parametrised versions of the NNLO splitting functions and mass threshold terms. The reader may verify that they correspond to those given in the top panel of table 15 of the second reference of [19] ($\mu_F = \mu_R$).

A.2 Streamlined interface

The program below generates exactly the same output as the previous example program, but this time using the streamlined interface introduced in section 8. It is to be found in `example_f90/tabulation_example_streamlined.f90`. A program with same interface and same output but in F77, is to be found in `example_f77/tabulation_example.f`.

```
program tabulation_example_streamlined
  use hoppet_v1
  !! if using LHAPDF, rename a couple of hoppet functions which
  !! would otherwise conflict with LHAPDF
  !use hoppet_v1, EvolvePDF_hoppet => EvolvePDF, InitPDF_hoppet => InitPDF
  implicit none
  real(dp) :: dy, ymax, dlnlnQ, Qmin, Qmax, muR_Q
  real(dp) :: asQ, Q0alphas, Q0pdf
  real(dp) :: mc,mb,mt
  integer  :: order, nloop
  !! holds information about the grid
  type(grid_def) :: grid, gdarray(4)
  !! hold results at some x, Q
  real(dp) :: Q, pdf_at_xQ(-6:6)
  real(dp), parameter :: heralhc_xvals(9) = &
       & (/1e-5_dp,1e-4_dp,1e-3_dp,1e-2_dp,0.1_dp,0.3_dp,0.5_dp,0.7_dp,0.9_dp/)
  integer  :: ix

  ! set up parameters for grid
  order = -6
  ymax  = 12.0_dp
  dy    = 0.1_dp

  ! set up the grid itself -- we use 4 nested subgrids
  call InitGridDef(gdarray(4),dy/27.0_dp,0.2_dp, order=order)
  call InitGridDef(gdarray(3),dy/9.0_dp,0.5_dp, order=order)
  call InitGridDef(gdarray(2),dy/3.0_dp,2.0_dp, order=order)
  call InitGridDef(gdarray(1),dy,       ymax  ,order=order)
  call InitGridDef(grid,gdarray(1:4),locked=.true.)

  ! Streamlined initialisation
  Qmin=1_dp
  Qmax=28000_dp
  dlnlnQ = dy/4.0_dp
  nloop = 3
  call hoppetStartExtended(ymax,dy,Qmin,Qmax,dlnlnQ,nloop,&
       &         order,factscheme_MSbar)

  ! Set heavy flavour scheme
```

```
  mc = 1.414213563_dp
  mb = 4.5_dp
  mt = 175.0_dp
  call hoppetSetVFN(mc, mb, mt)

    ! Set parameters of running coupling
  asQ = 0.35_dp
  Q0alphas = sqrt(2.0_dp)
  muR_Q = 1.0_dp
  Q0pdf = sqrt(2.0_dp) ! The initial evolution scale
  ! Normal evolution
  call hoppetEvolve(asQ, Q0alphas, nloop,muR_Q,&
       &       LHAsub, Q0pdf)

  ! Uncomment to perform  cached evolution
  ! call hoppetPreEvolve(asQ, Q0alphas, nloop,muR_Q,Q0pdf)
  ! call hoppetCachedEvolve(LHAsub)

  ! get the value of the tabulation at some point
  Q = 100.0_dp
  write(6,'(a)')
  write(6,'(a,f8.3,a)') "           Evaluating PDFs at Q = ",Q," GeV"
  write(6,'(a5,2a12,a14,a10,a12)') "x",&
       & "u-ubar","d-dbar","2(ubr+dbr)","c+cbar","gluon"
  do ix = 1, size(heralhc_xvals)
     call hoppetEval(heralhc_xvals(ix),Q,pdf_at_xQ)
     write(6,'(es7.1,5es12.4)') heralhc_xvals(ix), &
          &  pdf_at_xQ(2)-pdf_at_xQ(-2), &
          &  pdf_at_xQ(1)-pdf_at_xQ(-1), &
          &  2*(pdf_at_xQ(-1)+pdf_at_xQ(-2)), &
          &  (pdf_at_xQ(-4)+pdf_at_xQ(4)), &
          &  pdf_at_xQ(0)
  end do

contains

  subroutine LHAsub(x,Q,pdf)
    ! Same as in the previous example program
  end subroutine LHAsub

end program tabulation_example_streamlined
```

A.3 Other general-interface examples

To help better illustrate the usage of the general interface, additional examples are available in the `examples_f90/` and `benchmarking/` directories. One, `examples_f90/tabulation_example_2.f90`, summarised briefly below, illustrates how to distribute initialisation and evolution tasks across different parts of the program, and also illustrates how to verify sum rules. A second example for sum rules, `examples_f90/sumrules.f90`, also illustrates the use of evolution

without tabulation. The programs in the `benchmarking/` directory are the ones that have been used in benchmark PDF evolution comparisons [19] and in determining the accuracies and speeds shown in section 9.

One particular usage question that has arisen, relative to the example in appendix A.1, is what to do when a table (or other evolution components) need to be accessed from different subprograms, as would be the case in a typical global-fit code. The answer is that the derived-type variables that should persist outside a specific subroutine call must be placed in a `module` that is then accessed from each of the subprograms. The module might be

```
! common location for your table and other evolution components
module external_table_module
   use hoppet_v1
   implicit none
   type(pdf_table) :: table
   type(dglap_holder) :: dh
   type(running_coupling) :: coupling
   type(grid_def) :: grid  ! here for convenience, not strictly necessary
end module external_table_module
```

Here `table`, `dh` and `coupling` have all been placed in the module insofar as they are initialised/used in more than one location in the schematic subprograms below

```
subroutine A
   use external_table_module
   ! initialisation/allocation of grid, dh, table, coupling, ...
end subroutine A

subroutine B
   use external_table_module
   call PreEvolvePdfTable(table, Q0, dh, coupling)
end subroutine B

subroutine C
   use external_table_module
   ...
   call EvolvePdfTable(table,pdf0)
end subroutine C

subroutine D
   use external_table_module
   ...
   call EvalPdfTable_xQ(table, x, Q, pdf)
end subroutine D
```

A more detailed description of the above technique can be found in a second example program, called `tabulation_example_2.f90`, which is available in the `example_f90` directory. It generates exactly the same output as the previous example program, as well as additional output about truncated moments (sum rules), while providing an example of how to access a table from external procedures, as explained in section 7.2.

B HOPPET reference guide

In this section we present the HOPPET reference guide, a summary of the most important modules in the package with the corresponding description, both the streamlined interface, Table 4 and for the general interface, Table 5.

C Initialisation of grid quantities

As has been mentioned in section 4.2, there exist several ways of setting a grid quantity. In this Appendix we describe the most important methods for initialising a grid quantity, which we take to be a parton distribution.

There are a number of ways of setting a grid quantity. Suppose we have a subroutine

```
subroutine example_gluon(y,g)
  use types                !! defines "dp" (double precision) kind
  implicit none
  real(dp), intent(in)  :: y
  real(dp), intent(out) :: g
  real(dp) :: x

  x = exp(-y)
  g = 1.7_dp * x**(-0.1_dp) * (1-x)**5
end subroutine example_gluon
```

Then we can call

```
call InitGridQuantSub(grid,gluon,example_gluon)
```

to initialise `gluon` with a representation of the return value from the subroutine `example_gluon`.

An alternative way is to make use of functions `xValues` or `yValues` that respectively return the x or y values of all points on the grid:

```
real(dp), pointer :: gluon,xvals
call AllocGridQuant(grid,gluon)
call AllocGridQuant(grid,xvals)
xvals = xValues(grid)
gluon = 1.7_dp * xvals**(-0.1_dp) * (1-xvals)**5
deallocate(xvals)
```

Though more laborious insofar as one has to worry about some extra allocation and deallocation, it has the advantage that one no longer has to write a separate subroutine.

Finally, there is an option to initialise a multi-flavour PDF grid with a subroutine with the same format as `evolvePDF` from the `LHAPDF` library. This option works as follows:

```
real(dp),pointer :: pdf_set(:,:)
real(dp)         :: y,Q
real(dp)         :: pdf_at_y(-6:6)

Q=2   ! Initial scale

! Initialise with LHAPDF-like routine
```

STREAMLINED INTERFACE

METHOD	DESCRIPTION
Initialisation	
`hoppetStart(dy,nloop)`	Sets up a compound grid with spacing in $\ln 1/x$ of `dy` at small x, extending to $y = 12$ and numerical order `=` -6. The Q range for the tabulation will be 1 GeV $< Q <$ 28 TeV, `dlnlnQ=dy/4` and the factorisation scheme is $\overline{\rm MS}$
`hoppetStartExtended(ymax,dy,Qmin, Qmax,dlnlnQ,nloop,order,factscheme)`	More general initialisation
`hoppetSetFFN(fixed_nf)` `hoppetSetVFN(mc, mb, mt)`	Set heavy flavour scheme
`alphas = hoppetAlphaS(Q)`	Accessing the coupling
Normal evolution	
`hoppetEvolve(asQ,Q0alphas, nloop,muR_Q,LHAsub,Q0pdf)`	PDF evolution: specifies the coupling `asQ` at a scale `Q0alphas`, the number of loops for evol., `nloop`, the ratio (`muR_Q`) of ren. to fact. scales. the name of a subroutine `LHAsub` with an LHAPDF-like interface and the scale `Q0pdf` at which one starts the PDF evolution Note: `LHAsub` only called at scale `Q0pdf`
`hoppetEval(x,Q,f)`	On return, `f(-6:6)` contains all flavours of the PDF set (multiplied by x) at the given `x` and `Q` values
Cached evolution	
`hoppetPreEvolve(asQ,Q0alphas, nloop, muR_Q, Q0pdf)`	Preparation of the cached evolution
`hoppetCachedEvolve(LHAsub)`	Perform cached evolution with the initial condition at `Q0pdf` from a routine `LHAsub` with LHAPDF-like interface Notice `LHAsub` only called at scale `Q0pdf`
`hoppetEval(x,Q,f)`	On return, `f(-6:6)` contains all flavours of the PDF set (multiplied by x) at the given `x` and `Q` values [as for normal evolution]

Table 4: Reference guide for the streamlined interface. Note that HOPPET should be supplied with and returns parton densities multiplied by x.

GENERAL INTERFACE

TYPES	DESCRIPTION
`type(grid_def) :: grid`	$x-$space grid definition
`real(dp), pointer :: gluon(:)`	Holds a 'grid quantity' (e.g. gluon PDF)
`real(dp), pointer :: PDFset(:,:)`	Grid representation of a (13-flavour) PDF set
`type(grid_conv) :: Pgg`	Convolution operator (*i.e.* splitting function)
`type(split_mat) :: Pmat`	Splitting matrix (with full flavour structure)
`type(mass_threshold_mat) :: MTM_NNLO`	Heavy quark mass-threshold matrix
`type(dglap_holder) :: dglap_h`	DGLAP holder (i.e. all splitting and mass-threshold matrices)
`type(running_coupling) :: coupling`	Running coupling
`type(evln_operator) :: evop`	Evolution operator (linked list of split & mass-threshold matrices)
`type(pdf_table) :: table`	PDF set tabulated in x & Q

METHOD	DESCRIPTION
Initialisation	
`InitGridDef(grid,dy=0.1_dp,ymax=10.0_dp,order=3)`	Initialise a grid definition
`InitGridDef(grid,subgrids(:),locked=.true.)`	Combine subgrids into a single grid
`AllocGridQuant(grid,gluon)`	Allocate memory for a grid quantity (e.g. gluon PDF)
`InitGridQuant(grid,gluon,example_gluon_fn)`	Initialisation of a grid quantity (e.g. gluon PDF)
`AllocPDF(grid,pdfset)`	Allocate memory for a 13-flavour PDF set
`InitPDF_LHAPDF(grid,pdfset,LHAsub,Q)`	Initialisation of a (13-flavour) PDF set from LHAPDF type routine. Note: it only calls `LHAsub` at the scale `Q`
`InitGridConv(grid,Pgg,Pgg_func)` (*)	Initialisation of a convolution operator
`InitDglapHolder(grid, dglap_h, factscheme, nloop)` (*)	Initialisation of a `dglap_holder` type
`InitRunningCoupling(coupling [, alfas] [, Q] [, nloop] [, fixnf] [, quark_masses] )` (*)	Initialisation of a `running_coupling` type
`AllocPdfTable(grid, table, Qmin, Qmax [, dlnlnQ ] [, lnlnQ_order ] [, freeze_at_Qmin] )` (*)	Allocate space for a `pdf_table` type
Evaluation & manipulation	
`EvalGridQuant(grid,gluon,y)`	Evaluation of a grid quantity at $y = \ln 1/x$
`Pgg.conv.gluon`	Convolution of a splitting function with a (1-flav) PDF
`Pmat.conv.PDFset`	Convolution of a splitting matrix with a PDF set
`SetToConvolution(Pab,Pac,Pcb)`	Convolution of splitting functions, `Pab = Pac Pcb`
`EvalPdfTable_yQ(table, y, Q, pdf)`	Evaluate the 13 flavours, `pdf(-6:6)`, of a tabulated PDF set at y and Q
Evolution	
`CopyHumanPdfToEvln(nf_lcl, pdf_human, pdf_evln)`	Transform PDF set from `human` to `evln` representation
`GetPdfRep(pdfset)`	Check PDF set representation
`EvolvePDF(dglap_h, initial_pdfset, coupling, Q_init,Q_end ,[, muR_Q] [, nloop] [,du])`	Evolution of an initial condition for a PDF set at `Q_init` to `Q_end`
`EvolvePdfTable(table, Q0, initial_pdfset, dglap_h, coupling [, muR_Q] [, nloop] [, untie_nf])`	Fill a table starting from an initial condition, `initial_pdfset`, at the scale `Q0`

Table 5: Reference guide for the general interface. The upper table describes the main derived types defined in HOPPET. The lower table summarises some of the most relevant methods. Note that arguments between `[...]` are optional. For grid quantities and PDF sets the user must explicitly make memory allocation calls; in other cases (marked with a (*)), initialisation routines automatically allocate the memory. For all types, allocated memory may be freed with a call to the `Delete(...)` subroutine.

```
call AllocPDF(grid,pdf_set)
call InitGridQuantLHAPDF(grid, pdf_set, LHAsub, Q)

! Evaluate the multi-flavor pdf at y
pdf_at_y = EvalGridQuant(grid,pdf_set(:,-6:6),y)
```

where an example of the `LHAsub` subroutine can be found in section 5.1.1

D NNLO splitting functions

The exact NNLO splitting functions derived by Moch, Vermaseren and Vogt [22, 23] involve long (multi-page) expressions in terms of harmonic polylogarithms of up to weight 4. Very conveniently, refs. [22, 23] provide the expressions directly in terms of Fortran code. The harmonic polylogarithms can be evaluated using the `hplog` package of Gehrmann and Remiddi [48], a copy of which is included with the HOPPET package.

The initial integrations needed to create the `split_mat` objects for the exact NNLO splitting functions for the full range of n_f take of the order of minutes. Since currently there is no option of storing the splitting matrices in a file, this can be a bit bothersome. So instead, by default, the program uses the approximate, parametrised NNLO splitting functions also provided in [22, 23]. The parametrised splitting functions are guaranteed to be accurate to within 0.1% — in practice since they come in relatively suppressed by two powers of α_s, the impact on the evolution tends to be of the order of a 10^{-5} relative effect [19].

The user can choose whether to obtain the exact or parametrised NNLO splitting functions using the following calls (to be made before initialising the splitting matrices)

```
integer :: splitting_variant
call dglap_Set_nnlo_splitting(splitting_variant)
```

with the following variants defined (as integer parameters) in the module `dglap_choices`:

```
nnlo_splitting_exact
nnlo_splitting_param                    [default]
nnlo_splitting_Nfitav
nnlo_splitting_Nfiterr1
nnlo_splitting_Nfiterr2
```

The last 3 are the parametrisations based on fits to reduced moment information carried out in [35, 36]. Though at the time they represented a valuable (and much used) step on the way to full NNLO results, nowadays their interest is mainly historical.

Note that only for the `nnlo_splitting_exact` can the colour constants be varied (with the caveat about $d^{abc}d_{abc}$, as discussed in section 5.3.1). For the other options the NNLO splitting functions have been computed using the QCD values for the colour factors.

E Useful tips on Fortran 95

As Fortran 95's use in high-energy physics is not as widespread as that of other languages such as Fortran 77 and C++, it is useful to summarise some key novelties compared to Fortran 77, as well as some points that might otherwise cause confusion. For further information the reader is referred both to books about the language such as [49] and to web resources [50].

Free form. Most of the code in the HOPPET package is in free-form. The standard extension for free-form form files is `.f90`. There is no requirement to leave 6 blank spaces before every line and lines can consist of up to 132 characters. The other main difference relative to f77 fixed form is that to continue a line one must append an append an ampersand, `&`, to the line to be continued. One may optionally include an ampersand as the first non-space character of the continuation line.

For readability, many of the subprogram names in this documentation are written with capitals at the start of each word. Note however that free-form Fortran 95, like its fixed-form predecessors, is case insensitive.

Modules, and features relating to arrays. Fortran 95 allows one to package variables and subroutines into modules

```
module test_module
  implicit none
  integer :: some_integer
contains
  subroutine print_array(array)
    integer, intent(in) :: array(:) ! size is known, first element is 1
                                    ! intent(in) == array will not be changed
    integer             :: i, n
    n = size(array)
    do i = 1, n
      print *, i, array(i)
    end do
  end subroutine hello_world
end module test_module
```

The variable `some_integer` and the subroutine `print_array` are invisible to other routines unless they explicitly `use` the module as in the following example:

```
program test_program
  use test_module
  implicit none
  integer :: array1(5), array2(-2:2)
  integer :: i

  some_integer = 5   ! set the variable in test_module
  array1       = 0   ! set all elements of array1 to zero
  array2(-2:0) = 99  ! set elements 1..3 of array2 to equal to 3.
```

```
  array2(1:2)  = 2*array2(-1:0)  ! elements 1..2 equal twice elements -1..0

  print *, "Printing array 1"
  call print_array(array1)
  print *, "Printing array 2"
  call print_array(array2)
end program test_program
```

Constants can be assigned to arrays (`array1`) or array subsections (`array2(-2:0)`), arrays can be assigned to arrays of the same size (as is done for `array2(-2:0)`) and mathematical operations apply to each element of the array (as with the multiplication by 2).

When arrays are passed to function or subroutine that is defined in a `use`d module, information about the size of the array is passed along with the array itself. Note however that information about the lower bound is *not* passed, so that for both `array1` and `array2`, `print_array` will see arrays whose valid indices will run from $1 \ldots 5$. Thus the output from the program will be

```
 Printing array 1
 1 0
 2 0
 3 0
 4 0
 5 0
 Printing array 2
 1 99
 2 99
 3 99
 4 198
 5 198
```

If `print_array` wants `array` to have a different lower bound it must specify it in the declaration, for example

```
  integer, intent(in) :: array(-2:) ! size is known, first element is -2
```

While it may initially seem bizarre, there are good reasons for such behaviour (for example in allowing a subroutine to manipulate multiple arrays of the same size without having to worry about whether they all have the same lower bounds).

Dynamic memory allocation, pointers. One of the major additions of f95 compared to f77 is that of dynamic memory allocation, for example with pointers

```
  integer, pointer :: dynamic_array(:)
  allocate(dynamic_array(-6:6))
  ! .. work with it ..
  deallocate(dynamic_array)
```

This is fundamental to our ability to decide parameters of the PDF grid(s) at run-time. Pointers can be passed as arguments to subprograms. If the subprogram does not specify the `pointer` attribute for the dummy argument

```
subroutine xyz(dummy_array)
  integer, intent(in) :: dummy_array(:)
```

then everything behaves as if the argument were a normal array (e.g. the default lower bound is 1). Alternatively the subroutine can specify that it expects a pointer argument

```
subroutine xyz(dummy_pointer_array)
  integer, pointer :: dummy_pointer_array(:)
```

In this case the subroutine has the freedom to allocate and deallocate the array. Note also that because a pointer to the full array information is being passed, the lower bound of `dummy_pointer_array` is now the same as in the calling routine. Though this sounds like a technicality, it is important because a corollary is that a subroutine can allocate a dummy pointer array with bounds that are passed back to the calling subroutine (we need this for the flavour dimension of PDFs, whose lower bound is most naturally -6).

Note that in contrast to `C/C++` pointers, F95 pointers do not explicitly need to be dereferenced — in this respect they are more like `C++` *references*. To associate a pointer with an object, one uses the `=>` syntax:

```
  integer, target  :: target_object(10)
  integer, pointer :: pointer_to_object(:)
  pointer_to_object => target_object
  pointer_to_object(1:10) = 0        ! sets target_object(1:10)
```

One notes that the object that was pointed to had the `target` attribute — this is mandatory (unless the object is itself a pointer).

Derived types. Another feature of F95 that has been heavily used is that of derived types (analogous to C's `struct`):

```
  type pair
    integer first, second
  end type pair
```

Variables of this type can then be created and used as follows

```
  type(pair) :: pair_object, another_pair_object
  pair_object%first  = 1
  pair_object%second = 2
  another_pair_object = pair_object
  print *, another_pair_object%second
```

where one sees that the entirety of the object can be copied with the assignment (`=`) operator. Note that many of the derived types used in HOPPET contain pointers and when such a derived type object is copied, the copy's pointer just points to the same memory as the original object's pointer. This is sometimes what you want, but on other occasions will give unexpected behaviour: for example splitting function types are derived types containing pointers, so when you assign one splitting function object to another, they end up referring to the same memory, so if you multiply one of them by a constant, the other one will also be modified.

Operator overloading While assignment behaves more or less as expected by default with derived types (it can actually be modified if one wants to), other operators do not have default definitions. So if one wants to define, say, a multiplication of objects one may associate a function with a given operator, using an interface block:

```
module test_module
  interface operator(*)        ! provide access to dot_pairs through
    module procedure dot_pairs ! the normal multiplication symbol
  end interface
  interface operator(.dot.)    ! provide access to dot_pairs through
    module procedure dot_pairs ! a specially named operator
  end interface
contains
  integer function dot_pairs(pair1, pair2)
    type(pair), intent(in) :: pair1, pair2
    dot_pairs = pair1%first*pair2%first + pair1%second*pair2%second
  end function dot_pairs
end module
```

given which we can then write

```
  integer    :: i
  type(pair) :: pair1, pair2
  [... some code to set up pair values ...]
  ! now multiply them
  i = pair1 * pair2
  i = pair1 .dot. pair2  ! equivalent to previous statement
```

Since the multiplication operator (`*`) already exists for all the default types, by defining it for a new type we have *overloaded* it. Note that there are some subtleties with precedences of user-defined operators: operators (like `*`) that already exist have the same precedence as they have is usual operators; operators that do not exist by default (`.dot`) have the lowest possible preference, so, given the above definitions,

```
  i = 2 + pair1 * pair2       ! legal
  i = 2 + pair1 .dot. pair2   ! illegal, means: (2+pair1).dot.pair2
  i = 2 + (pair1 .dot. pair2) ! legal
```

where the second line is illegal because we have not defined any operator for adding an integer and a pair. Similarly care is needed when using the HOPPET's operator `.conv.`.

Floating-point precision: A final point concerns floating-point variable types. Throughout we have used definitions such as

```
  real(dp), pointer :: pdf(:,:)
```

and written numbers with a trailing `_dp`

```
  param = 1.7_dp
```

Here `dp` is an integer parameter (defined in the `types` module and accessible also through the `hoppet_v1` module), which specifies the `kind` of real that we want to define, specifically double precision. We could also have written `double precision` everywhere, but this is

less compact, and the use of a kind parameter has the advantage that we can just modify its definition in one point in the program and the precision will be modified everywhere. (Well, almost, since some special functions are written in Fortran 77 using `double precision` declarations and do their numerics based on the assumption that that truly is the type they're dealing with).

A word of caution: `1_dp` is *not* a double-precision number, but rather an integer (e.g. on many compilers `dp=8`, indicating a number with 8 bytes of storage, so that `1.0_dp` is an 8-byte floating-point number and `1_dp` is an 8-byte integer). Therefore trying to pass `1_dp` as the value for a double-precision dummy argument in a subprogram call will lead to a compile-time error.

Optional and keyword arguments A feature of F95 that helps simplify user interfaces is that of optional and keyword arguments. Suppose we have

```
subroutine hello(name, prefix, count)
  character(len=*),  intent(in)  :: name, prefix
  integer, optional, intent(in)  :: count
end subroutine hello
```

Here the `count` argument is `optional` meaning that it need not be supplied — if it is absent the subroutine is should behave sensibly all the same. Thus one can call the subroutine as

```
call hello(name, prefix)
call hello(name, prefix, count)
```

Keyword arguments are useful if one doesn't want to remember the exact order of a long list of arguments (or if one wants to specify just one of several optional arguments). For example

```
call hello(name=name, prefix=prefix)
call hello(prefix=prefix, name=name)
```

will do the same thing.

References

[1] M. Botje, QCDNUM, `http://www.nikhef.nl/~h24/qcdnum/` .

[2] L. Schoeffel, Nucl. Instrum. Meth. A **423** (1999) 439. See also `http://www.desy.de/~schoffel/L_qcd98.html`, `http://www-spht.cea.fr/pisp/gelis/Soft/DGLAP/index.html`

[3] A. Vogt, Comput. Phys. Commun. **170** (2005) 65 [arXiv:hep-ph/0408244].

[4] C. Pascaud and F. Zomer, arXiv:hep-ph/0104013.

[5] S. Weinzierl, Comput. Phys. Commun. **148** (2002) 314 [arXiv:hep-ph/0203112]; M. Roth and S. Weinzierl, Phys. Lett. B **590** (2004) 190 [arXiv:hep-ph/0403200].

[6] A. Cafarella and C. Coriano, Comput. Phys. Commun. **160** (2004) 213 [arXiv:hep-ph/0311313]; A. Cafarella, C. Coriano' and M. Guzzi, Nucl. Phys. B **748** (2006) 253 [arXiv:hep-ph/0512358]; A. Cafarella, C. Coriano and M. Guzzi, arXiv:0803.0462 [hep-ph].

[7] M. Guzzi, Ph.D. Thesis, Lecce University, 2006 [hep-ph/0612355].

[8] L. Del Debbio, S. Forte, J. I. Latorre, A. Piccione and J. Rojo [NNPDF Collaboration], JHEP **0703** (2007) 039 [arXiv:hep-ph/0701127].

[9] D. A. Kosower, Nucl. Phys. B **506** (1997) 439 [arXiv:hep-ph/9706213].

[10] P. G. Ratcliffe, Phys. Rev. D **63**, 116004 (2001) [arXiv:hep-ph/0012376].

[11] V.N. Gribov and L.N. Lipatov, Sov. J. Nucl. Phys. **15** (1972) 438; G. Altarelli and G. Parisi, Nucl. Phys. B **126** (1977) 298; Yu.L. Dokshitzer, Sov. Phys. JETP **46** (1977) 641.

[12] J. Pumplin, D. R. Stump, J. Huston, H. L. Lai, P. Nadolsky and W. K. Tung, JHEP **0207**, 012 (2002) [arXiv:hep-ph/0201195].

[13] A. D. Martin, R. G. Roberts, W. J. Stirling and R. S. Thorne, Phys. Lett. B **531** (2002) 216 [arXiv:hep-ph/0201127].

[14] M. Dasgupta and G. P. Salam, Eur. Phys. J. C **24** (2002) 213 [arXiv:hep-ph/0110213]; JHEP **0208** (2002) 032 [arXiv:hep-ph/0208073].

[15] A. Banfi, G. P. Salam and G. Zanderighi, JHEP **0503**, 073 (2005) [arXiv:hep-ph/0407286]; JHEP **0408**, 062 (2004) [arXiv:hep-ph/0407287].

[16] M. Ciafaloni, D. Colferai, G. P. Salam and A. M. Stasto, Phys. Rev. D **68**, 114003 (2003) [arXiv:hep-ph/0307188].

[17] T. Carli, G. P. Salam and F. Siegert, :hep-ph/0510324; T. Carli, D. Clements, *et al.*, in preparation.

[18] A. Banfi, G. P. Salam and G. Zanderighi, JHEP **0707** (2007) 026 [arXiv:0704.2999 [hep-ph]].

[19] W. Giele *et al.*, "Les Houches 2001, the QCD/SM working group: Summary report," hep-ph/0204316, section 1.3;
M. Dittmar *et al.*, "Parton distributions: Summary report for the HERA-LHC workshop," hep-ph/0511119, section 4.4.

[20] W. Giele and M. R. Whalley, `http://hepforge.cedar.ac.uk/lhapdf/`

[21] W. Furmanski and R. Petronzio, Phys. Lett. B **97** (1980) 437; G. Curci, W. Furmanski and R. Petronzio, Nucl. Phys. B **175** (1980) 27.

[22] S. Moch, J. A. M. Vermaseren and A. Vogt, Nucl. Phys. B **688** (2004) 101 [arXiv:hep-ph/0403192].

[23] A. Vogt, S. Moch and J. A. M. Vermaseren, Nucl. Phys. B **691** (2004) 129 [arXiv:hep-ph/0404111].

[24] R. Mertig and W. L. van Neerven, Z. Phys. C **70** (1996) 637 [arXiv:hep-ph/9506451].

[25] W. Vogelsang, Nucl. Phys. B **475** (1996) 47 [arXiv:hep-ph/9603366].

[26] M. Stratmann and W. Vogelsang, Nucl. Phys. B **496** (1997) 41 [arXiv:hep-ph/9612250].

[27] Yu. L. Dokshitzer, G. Marchesini and G. P. Salam, Phys. Lett. B **634**, 504 (2006) [arXiv:hep-ph/0511302].

[28] A. Mitov, S. Moch and A. Vogt, Phys. Lett. B **638** (2006) 61 [arXiv:hep-ph/0604053].

[29] B. Basso and G. P. Korchemsky, Nucl. Phys. B **775** (2007) 1 [arXiv:hep-th/0612247].

[30] Yu. L. Dokshitzer and G. Marchesini, Phys. Lett. B **646** (2007) 189 [arXiv:hep-th/0612248].

[31] M. Beccaria, Yu. L. Dokshitzer and G. Marchesini, Phys. Lett. B **652** (2007) 194 [arXiv:0705.2639 [hep-th]].

[32] M. Buza, Y. Matiounine, J. Smith, R. Migneron and W. L. van Neerven, Nucl. Phys. B **472**, 611 (1996) [arXiv:hep-ph/9601302];
M. Buza, Y. Matiounine, J. Smith and W. L. van Neerven, Eur. Phys. J. C **1**, 301 (1998) [arXiv:hep-ph/9612398].

[33] K. G. Chetyrkin, B. A. Kniehl and M. Steinhauser, Phys. Rev. Lett. **79**, 2184 (1997) [arXiv:hep-ph/9706430].

[34] A. Sherstnev and R. S. Thorne, arXiv:0807.2132 [hep-ph].

[35] W. L. van Neerven and A. Vogt, Nucl. Phys. B **568** (2000) 263 [arXiv:hep-ph/9907472].

[36] W. L. van Neerven and A. Vogt, Nucl. Phys. B **588** (2000) 345 [arXiv:hep-ph/0006154].

[37] Press *et al.*, *Numerical Recipes in Fortran 90*, Cambridge University Press, 1996.

[38] A. Vogt, private communication.

[39] C. D. White and R. S. Thorne, Eur. Phys. J. C **45** (2006) 179 [arXiv:hep-ph/0507244].

[40] S. Bethke, Prog. Part. Nucl. Phys. **58**, 351 (2007) [arXiv:hep-ex/0606035].

[41] D. de Florian, R. Sassot and M. Stratmann, Phys. Rev. D **76**, 074033 (2007) [arXiv:0707.1506 [hep-ph]].

[42] G. Corcella and L. Magnea, Phys. Rev. D **72** (2005) 074017 [arXiv:hep-ph/0506278].

[43] A. D. Martin, W. J. Stirling, R. S. Thorne and G. Watt, Phys. Lett. B **652**, 292 (2007) [arXiv:0706.0459 [hep-ph]].

[44] W. K. Tung, H. L. Lai, A. Belyaev, J. Pumplin, D. Stump and C. P. Yuan, JHEP **0702**, 053 (2007) [arXiv:hep-ph/0611254].

[45] A. D. Martin, R. G. Roberts, W. J. Stirling and R. S. Thorne, Eur. Phys. J. C **39**, 155 (2005) [arXiv:hep-ph/0411040].

[46] M. Ciafaloni, P. Ciafaloni and D. Comelli, Phys. Rev. Lett. **84**, 4810 (2000) [arXiv:hep-ph/0001142].

[47] P. Ciafaloni and D. Comelli, JHEP **0511**, 022 (2005) [arXiv:hep-ph/0505047].

[48] T. Gehrmann and E. Remiddi, Comput. Phys. Commun. **144** (2002) 200.

[49] M. Metcalf and J. Reid, *Fortran 90/95 Explained*, Oxford University Press, 1996.

[50] Many introductions and tutorials about fortran 90 may be found at `http://dmoz.org/Computers/Programming/Languages/Fortran/Tutorials/Fortran_90_and_95/`