ColorFull: a C++ library for calculations in SU(Nc) color space

Malin Sjödahl
Department of Astronomy and Theoretical Physics, Lund University, Sölvegatan 14A, 22362 Lund, Sweden

Received: 22 December 2014 / Accepted: 17 April 2015 / Published online: 29 May 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract ColorFull, a C++ package for treating QCD color structure, is presented. ColorFull, which utilizes the trace basis approach, is intended for interfacing with event generators, but can also be used as a stand-alone package for squaring QCD amplitudes, calculating interferences, and describing the effect of gluon emission and gluon exchange.

Contents

1 Introduction ........................................ 1
2 Color space ........................................ 2
3 Trace bases ........................................ 2
4 Computational strategy ............................. 4
5 ColorFull at a glance ................................ 4
5.1 Technical overview ............................... 4
6 Stand-alone usage ................................ 5
7 Interfacing to Herwig via matchbox .............. 7
8 ColorFull code ...................................... 7
  8.1 Operators ......................................... 7
  8.2 Classes for polynomials ......................... 7
    8.2.1 Monomial .................................. 8
    8.2.2 Polynomial ................................ 9
    8.2.3 Poly_vec .................................. 9
    8.2.4 Poly_matr ................................ 9
  8.3 Classes containing the color structure ........ 9
    8.3.1 Quark_line ................................ 9
    8.3.2 Col_str ................................... 10
    8.3.3 Col_amp .................................. 10
  8.4 Col_functions: a library class ................. 10
    8.4.1 Numerical evaluation ....................... 11
    8.4.2 Leading $N_c$ evaluation .................. 11
    8.4.3 Scalar products ............................ 11
    8.4.4 Gluon exchange and gluon emission ........ 11
  8.5 Classes for color bases ....................... 11
    8.5.1 Col_basis .................................. 12
  8.5.2 Trace_type_basis ............................ 12
  8.5.3 Trace_basis ................................ 12
  8.5.4 Tree_level_gluon_basis ..................... 12
  8.5.5 Orthogonal_basis ........................... 13
9 Validation ........................................... 13
10 Conclusion and outlook ......................... 13
Appendix A: Class member functions .............. 13
References .......................................... 41

1 Introduction

The description of QCD color structure in the presence of many external colored partons is a field of increased importance. Some methods for performing automatic color summations of fully contracted vacuum bubbles, for example as implemented in FeynCalc [1], in the C program COLOR [2], or as presented in [3], have been around for a while, and recently a more flexible Mathematica package, ColorMath [4], allowing color structures with any number of open indices, has been published. Yet other general purpose event generator codes, such as MadGraph [5], have separate built in routines for dealing with the color structure.

In the present paper a stand-alone C++ code, ColorFull, designed for dealing with color contraction using color bases is presented. ColorFull is written with interfacing to event generators in mind, and is currently interfaced to Herwig++(2.7) [6,7], but can also be used as a stand-alone package for investigations in color space.

ColorFull is based on trace bases [8–16], where the basis vectors are given by (products of) closed and open quark-lines, but the code also offers functionality for reading in and treating any (orthogonal) basis for color space, such as multiplet bases [17,18].

The intent of this paper is to convey the underlying idea of ColorFull. For full technical details we refer to the online

---

1 ColorFull can be downloaded from http://colorfull.hepforge.org/.
Doxygen documentation\textsuperscript{2}. To set the stage, a brief introduction to the QCD color space is given in Sect. 2 and the trace basis approach is presented in Sect. 3. Following this, some remarks about the computational strategy are made in Sect. 4 and the key design features are presented in Sect. 5, whereas examples of stand-alone usage are given in Sect. 6, and the interface to Herwig++ is commented upon in Sect. 7. The following section, Sect. 8, describes the classes of ColorFull and code validation is discussed in Sect. 9. Finally some concluding remarks are made in Sect. 10.

2 Color space

Apart from four-gluon vertices for which the color structure can be rewritten in terms of (one-gluon contracted) triple-gluon vertices, the QCD Lagrangian contains

\[
\text{quark-gluon vertices, } \quad i \quad \begin{array}{c} a \to b \to c \end{array} \quad j = (\epsilon^a)^j, \quad (2.1)
\]

and

\[
\text{triple-gluon vertices, } \quad \begin{array}{c} a \to b \to c \end{array} \quad i \quad \begin{array}{c} d \to e \to f \end{array} = i f^{abc}, \quad (2.2)
\]

where we follow the convention of reading the fully anti-symmetric structure constant indices in counter clock-wise order. The color structure of any amplitude, tree-level or beyond, pure QCD or not, can thus be expressed in terms of these objects alone. For observables we are – as QCD is confining – only interested in color summed/averaged quantities. Letting \( c_1 \) denote the color structure of the amplitude under consideration, we are thus interested in \( |c_1|^2 \) where the scalar product\textsuperscript{3} is given by summing over all external color indices, i.e.,

\[
(c_1 | c_2) = \sum_{a_1, a_2, \ldots} c_1^{a_1 a_2 \ldots} c_2^{a_1 a_2 \ldots} \quad (2.3)
\]

with \( a_i = 1, \ldots, N_c \) if parton \( i \) is a quark or an anti-quark and \( a_i = 1, \ldots, N_c^2 - 1 \) if parton \( i \) is a gluon.

Clearly, in any QCD calculation, the color amplitudes, \( c_1 \) and \( c_2 \), may be kept as they are, with color structure read off from the contributing Feynman diagrams. Alternatively – and this is likely to be the preferred solutions for more than a few partons – they may be decomposed into a color basis (spanning set), such as a trace basis [8–16], a color flow basis [19] or a multiplet basis [17,18].

3 Trace bases

One way of organizing calculations in color space is to use trace bases [8–16]. To see that this is always possible, we note that the triple-gluon vertex can be expressed as

\[
i f^{abc} = \frac{1}{T_R} \left[ \begin{array}{c} \epsilon^a \to \epsilon^b \to \epsilon^c \end{array} \right] = \frac{1}{T_R} \left[ (\epsilon^a)^j \epsilon^b \epsilon^c - (\epsilon^b)^j \epsilon^a \epsilon^c \right]
\]

where \( T_R \) is the normalization of the SU(\( N_c \)) generators, \( \text{tr}(\epsilon^a \epsilon^b \epsilon^c) = T_R \delta^{abc} \), commonly taken to be 1/2 or 1.

Using this relation on every triple-gluon vertex in any amplitude results in general in a sum of products of (connected) traces over SU(\( N_c \)) generators and open quark-lines. More specifically, there is one open quark-line for every incoming quark/outgoing anti-quark and outgoing quark/incoming anti-quark. (Note that, from a color perspective outgoing anti-quarks are equivalent to incoming quarks; we will here refer to them collectively as quarks. Similarly outgoing quarks are equivalent to incoming anti-quarks, and will be referred to as anti-quarks.)

To further simplify the color structure, we may contract every internal gluon propagator (which after application of Eq. (3.1) connects quark-lines) using the Fierz (complete-ness) relation

\[
\begin{array}{c} a \to \epsilon^a \to \epsilon^b \to \epsilon^c \end{array} = T_R \left[ \begin{array}{c} \epsilon^a \to \epsilon^b \to \epsilon^c \end{array} - \frac{1}{N_c} \right]
\]

From this we see that every amplitude in QCD, at any order, may be expressed as a sum of products of open and closed quark-lines, where all gluon indices are external indices. The set of all such products of quark-lines can thus be used as a spanning set for any QCD process, for example, for one \( q \bar{q} \)-pair and seven gluons, we may have color structures of form

\[
\begin{array}{c} a \to \epsilon^a \to \epsilon^b \to \epsilon^c \end{array}
\]

for all possible gluon permutations, as well as color structures with more or fewer traces. We here refer to this type of basis as a trace basis, although we remark that when the number of gluons, \( N_g \), plus the number of \( q \bar{q} \)-pairs, \( N_q \), exceeds \( N_c \), this spanning set is overcomplete, and hence strictly speaking not a basis. For \( N_g + N_q \leq N_c \), the bases are not overcomplete, but they are still non-orthogonal, having non-diagonal scalar products being suppressed by powers of

\textsuperscript{2} The automatically generated Doxygen documentation is available at http://colorfull.hepforge.org/doxygen.

\textsuperscript{3} It is not hard to prove that this actually is a scalar product.
$N_c$. Only in the $N_c \to \infty$ limit, are these bases minimal and orthogonal.

As a simple, but non-trivial example, we may consider the basis needed for $q_1\overline{q}_2 \to g_3 g_4$. The basis, which is constructed by connecting quarks and gluons in all allowed ways \[16\] is given by \[4\]

\begin{align*}
\mathcal{V}^0_{q_1,q_2,g_3,g_4} &= (t^6 t^{\frac{1}{2}})^{q_1} = \begin{array}{c}
q_1 \\
g_3 \\
g_4 \\
q_2
\end{array} \\
\mathcal{V}^1_{q_1,q_2,g_3,g_4} &= (t^6 t^{\frac{1}{2}})^{q_1} = \begin{array}{c}
q_1 \\
g_3 \\
g_4 \\
q_2
\end{array} \\
\mathcal{V}^2_{q_1,q_2,g_3,g_4} &= \delta^{q_1}_{q_2} \text{tr}(t^6 t^{\frac{1}{2}}) = \begin{array}{c}
q_1 \\
g_3 \\
g_4 \\
q_2
\end{array}
\end{align*}
(3.4)

To all orders in perturbation theory in the $N_c \to \infty$ limit, one can prove that the number of basis vectors can be found using the recursion relation \[17\]

\begin{align*}
N_{\text{vec}}[N_q, Ng] &= N_{\text{vec}}[N_q, Ng - 1](Ng - 1 + N_q) \\
&\quad + N_{\text{vec}}[N_q, Ng - 2](Ng - 1), \\
&\quad (3.5)
\end{align*}

where

\begin{align*}
N_{\text{vec}}[N_q, 0] &= N_q!, \\
&N_{\text{vec}}[N_q, 1] = N_q N_q!.
\end{align*}
(3.6)

In the gluon-only case, at tree-level, the only color structures that can appear are traces over generators, meaning that a general tree-level gluon amplitude can be decomposed as

\begin{align*}
\mathcal{M}(g_1, g_2, \ldots, g_n) &= \sum_{\sigma \in S_{g-1}} \text{tr}(t^{\sigma_1} t^{\sigma_2} \ldots t^{\sigma_n}) A(\sigma) \\
&= \sum_{\sigma \in S_{g-1}} \begin{array}{c}
g_1 \\
g_2 \\
\ldots \\
g_n
\end{array} A(\sigma).
\end{align*}
(3.7)

That only fully connected color structures enter in tree-level gluon amplitudes can easily be understood from the decomposition of Feynman diagrams into basis vectors; upon application of Eq. (3.1) all external gluons remain attached to a quark-line, and – while contracting internal gluons using the Fierz identity, Eq. (3.2) – they remain connected to the same quark-line, as the color suppressed terms cancel each other out. (This can be proved by a short calculation.) The same cancellation appears for gluon exchange between a quark and a gluon, meaning that also tree-level color structures for one $q\overline{q}$-pair and $N_g$ gluons must be of the “fully connected” form of a trace that has been cut open, an open quark-line,

\begin{align*}
\mathcal{M}(q_1, g_3, \ldots, g_n, \overline{q}_2) &= \sum_{\sigma \in S_{g-1}} (t^{\sigma_1} t^{\sigma_2} \ldots t^{\sigma_n})^{q_1} \overline{q}_2 A(\sigma) \\
&= \sum_{\sigma \in S_{g-1}} \begin{array}{c}
g_1 \\
g_2 \\
\ldots \\
g_n
\end{array} A(\sigma),
\end{align*}
(3.8)

i.e., only the first two basis vectors in Eq. (3.4) are needed. However, when the Fierz identity is applied directly to a gluon exchange between quarks, as in Eq. (3.2), both terms do appear, and color structures with up to $N_g$ disconnected quark-lines may appear even at tree-level.

Starting from a trace basis tree-level color structure, for example a single trace over gluons, and exchanging a gluon between two partons may split off a disconnected color structure, such as in

\begin{align*}
\begin{array}{c}
g_1 \\
g_2 \\
g_3 \\
g_4
\end{array} &= -\tau_k \\
\begin{array}{c}
g_1 \\
g_2 \\
\ldots \\
g_n
\end{array}
\end{align*}
(3.9)

Thus, counting to $l_g$ additional gluon exchanges (on top of a tree-level diagram), the color structures can not consist of more than max$(1, N_g) + l_g$ open and closed quark-lines, two in the above case. In general, when any Feynman diagram is decomposed into a trace basis, there can be at most $N_q + [N_g/2]$ quark-lines, since all gluons may be disconnected from the quarks, but no gluon can stand alone in a trace, giving the factor $[N_g/2]$.

For NLO color structures having a quark-loop in the Feynman diagram, the quark-loop is necessarily connected to the remaining color structure via at least one gluon exchange, unless the Feynman diagram consists of only gluons attached to one quark-loop, in which case the trace basis decomposition is trivial. In general it may be connected to more tree-level color structures, for example of form

\begin{align*}
\begin{array}{c}
g_1 \\
\ldots \\
g_n
\end{array} &= -\tau_k
\end{align*}
(3.10)

where the gray blobs denote any tree-level color structures. For each color structure connected to the quark-loop (gray blob), contracting the connecting gluon gives a term where the quark-loop is disconnected if the connecting gluon goes to a quark, the second term in
If the quark-loop is not connected to any other color structure (there is no remaining gray blob in the equation above), the resulting color structure thus contains one more single trace meaning that, decomposed in the trace basis, we get one more quark-line than at tree-level. (For purely gluonic processes, no additional trace will appear.) The same argument can be applied repeatedly if the Feynman diagram contains more than one closed quark-loop, giving (similar to the gluon case) no more than \( \max(N_q, 1) + l_q \) quark-lines for \( l_q \) quark-loops.

Finally, we note that, if, on top of the quark-loops, there are also \( l_g \) loops from gluon exchanges in the Feynman diagram, these may contribute with up to one additional quark-line each (such as in Eq. (3.9)), completing the argument that for \( n_1 = l_q + l_g \) loops, a general color structure in pure QCD can be written in terms of sums of products of closed and open quark-lines, where each product contains at most \( \min[\max(1, N_q) + n_1, N_q + [N_g/2]] \) traces and open quark-lines.

4 Computational strategy

The ColorFull strategy for treating color space is based on the above observations, i.e.,

- For given external particles (quarks, anti-quarks and gluons), we may always decompose the color space into a linear combination of closed and open quark-lines, as described in Sect. 3. (Other color bases may be expressed in terms of linear combinations of these terms.)
- We can always evaluate scalar products by first replacing triple-gluon vertices using Eq. (3.1), then removing all gluon propagators using Eq. (3.2), and finally contracting the remaining product of quark-delta functions.

The above outlined procedure for calculating scalar products has the advantages of being conceptually easy, and of covering all contractions occurring in QCD. It has the disadvantage of potentially giving a very large number of terms, since every replacement of a structure using Eq. (3.1) may double the number of terms, and similarly, so does naive application of the Fierz identity, Eq. (3.2).

In order to mitigate this potential explosion of terms upon index contraction, the ColorFull procedure for scalar product contraction is:

1. Contract all quark-ends, giving a sum of products of closed quark-lines.
2. On the individual quark-lines, contract all neighboring gluons (giving a factor \( C_F \)) and all next to neighboring gluons, giving a factor \(-T_R/N_c\) each, as only the color suppressed term in the Fierz identity, Eq. (3.2), survives. Also, contract traces of two gluons, \( \text{tr}(t^t_1 t^t_2) = g^{t_1 t_2} T_R \).
3. Look for gluons to contract within the quark-lines. Each such contraction may give rise to two terms, but at least the new traces tend to be shorter.
4. Look for gluon index contractions between the quark-lines.

In this way all color indices can be iteratively contracted. To further speed up calculations, ColorFull can use memoization to save calculated topologies, meaning that contractions that differ only by a relabeling of indices are performed only once. This significantly speeds up the calculations.

5 ColorFull at a glance

ColorFull is designed to handle the contraction of QCD color indices, to decompose the QCD color space using (trace) basis vectors, and to describe the effect of gluon exchange and gluon emission. In particular, ColorFull can, for arbitrary \( N_c \):

- Square any QCD amplitude and calculate any interference term.
- Create a trace basis for any number of quarks and gluons, and to arbitrary order.
- Read in and write out color bases, including non-trace bases.
- Calculate scalar product matrices, i.e., the matrices of scalar products between the basis vectors, write these out and read them in again.
- Describe the effect of gluon exchange, including calculating the color soft anomalous dimension matrices.
- Describe the effect of gluon emission.

ColorFull can also be interfaced to Herwig++ (\( \geq 2.7 \)) via Matchbox [20] and can thus easily be used for event generation with Herwig++.

5.1 Technical overview

This section is intended to give an overview of ColorFull. For examples, the reader is referred to Sect. 6, for overviews of the classes to Sect. 8, and for details, the tables in the Appendix A, as well as the online Doxygen documentation.
The classes intended for the user, Trace, Tree_level_gluon_basis and the first two via the class Trace_type_basis. Col_basis, Col_str, color amplitudes, i.e., a linear combination of emission and exchange, ColorFull.

Throughout the ColorFull documentation terms of form constant×\(N_c^{pow}T_R^{pow}C_F^{pow}\), will be referred to as monomials, despite the possible occurrence of negative powers. Similarly sums of such terms will be referred to as polynomials.

As contraction of indices gives rise to polynomials in \(N_c\), \(T_R\) and \(C_F\), ColorFull by necessity needs minimal classes for dealing with such polynomials. This is implemented in the classes Monomial (a single term in a polynomial), Polynomial (a sum of Monomials), Poly_vec (a vector of Polynomials) and Poly_matr (a matrix of Polynomials).

For the color structure itself, ColorFull uses the class Quark_line for treating an individual closed or open quark-line, a class Col_str for treating a product of Quark_lines and a class Col_amp for treating a general color amplitudes, i.e., a linear combination of Col_strs.

For performing numerical evaluations, taking the leading \(N_c\) limit, evaluating scalar products, and describing gluon emission and exchange, ColorFull has a library class Col_functions.

Finally, ColorFull has classes for describing color bases. The classes intended for the user, Trace_basis, Tree_level_gluon_basis and Orthogonal_basis, are derived from the base class Col_basis (in the case of the first two via the class Trace_type_basis).

All ColorFull classes are listed in Table 1 according to dependencies, meaning that each class only depends on classes listed above it. The next section will give an introduction to using ColorFull in stand-alone mode.

### Table 1 Below is a complete list of the ColorFull classes, ordered by dependence, such that, a certain class only depends on classes standing above it in the list.

| Class                      | Functionality                                                                 |
|----------------------------|------------------------------------------------------------------------------|
| Monomial                   | Class for containing a single term of form \(N_c^{pow}T_R^{pow}C_F^{pow}\) \times int_part \times cnun_part and associated functions. |
| Polynomial                 | Class for containing a sum of Monomials and associated functions.              |
| Poly_vec                   | Class for containing a vector of Polynomials and associated functions.        |
| Poly_matr                  | Class for containing a matrix of Polynomials and associated functions.        |
| Quark_line                 | Class for containing a single closed or open quark-line multiplying a Polynomial and associated functions. For example a polynomial \((N_c^2-1)\) may multiply the Quark_lines, \(T_R^{pow}C_F^{pow}\), in total giving the Col_str \(\sim (Nc2-1) [(1,2)(3,4)]\). |
| Col_str                    | Class for containing a product of Quark_lines multiplying a Polynomial, and associated functions. For example a term \((1,2,3,4)+(1,2)(3,4)\) or of the form \((T_R^{pow}C_F^{pow})_t \sim [1,3,4,2] \times \) times a Polynomial. |
| Col_amp                    | Class for containing a sum of Col_strs such as \(tr[T_R^{pow}C_F^{pow}]+tr[T_R^{pow}]tr[T_R^{pow}]+tr[T_R^{pow}]\) \(\sim (1,2,3,4)+(1,2)(3,4)\) and associated functions. |
| Col_functions              | Library class containing functions for performing numerical evaluations, taking the leading \(N_c\) limit, evaluating scalar products, and describing gluon emission and exchange. |
| Col_basis                  | Base class for all basis classes, see below.                                  |
| Trace_type_basis           | Base class for Trace_basis and Tree_level_gluon_basis.                        |
| Trace_basis                | Class for creating and using trace bases. Perhaps the most important basis class. |
| Tree_level_gluon_basis     | Class for bases needed for tree-level gluon calculations, where each basis vector is a single trace plus an implicit complex conjugate. |
| Orthogonal_basis          | Class for utilizing the benefits of orthogonal bases such as multiplet bases [17,18]. |

6 Stand-alone usage

ColorFull is mainly designed to deal with relatively large color spaces where it is advantageous to use bases in which color structures coming from Feynman diagrams – or alternative recursive strategies – are decomposed.

For trace bases, the bases may be automatically created by ColorFull. For example, a basis for 1 \(q\bar{q}\)-pair, 3 gluons and 0 loops (in pure QCD) can be created using

\[
\text{Trace_basis MyBasis(1,3,0)}; \\
\text{MyBasis.write_out Col_basis} \text{("ColorResults/MyBasis");}
\]

The last argument is provided to avoid carrying around basis vectors for which the kinematic factors vanish to a certain order in perturbation theory. It can be skipped upon which an all order basis vector is created.

To view the resulting basis, it can be written out to a stream (cout) or to a file, either with default filename (no argument) or a user supplied name,
resulting in

\[
\begin{align*}
0 & \quad \left\{ [1, 3, 4, 5, 2] \right\} \\
1 & \quad \left\{ [1, 3, 5, 4, 2] \right\} \\
2 & \quad \left\{ [1, 4, 3, 5, 2] \right\} \\
3 & \quad \left\{ [1, 4, 5, 3, 2] \right\} \\
4 & \quad \left\{ [1, 5, 3, 4, 2] \right\} \\
5 & \quad \left\{ [1, 5, 4, 3, 2] \right\}
\end{align*}
\]  

where, for example,

\[
\{1, 3, 4, 5, 2\} = (e^{3}e^{4}e^{5})q_{1} = q_{1} \quad q_{2} \quad q_{3} \quad q_{4} \quad q_{5}.
\]  

If no argument is supplied to \texttt{MyBasis.write_out_Col\_basis()}, the basis is written out to a file with a default filename, in this case \texttt{CF\_TB\_q\_1\_g\_3}. The bases can thus be written out and saved for future purposes. For reading in a basis \texttt{MyBasis.read\_in\_Col\_basis(“path/to/filename“)} can be used.

The option of reading in bases is particularly useful for the \texttt{Orthogonal\_basis} class. Presently \texttt{ColorFull} cannot automatically create orthogonal multiplet bases, such as in [17], but externally created (orthogonal) bases, with basis vectors expressed in terms of sums of products of traces, may be read in and used. For the \texttt{Orthogonal\_basis} class, the orthogonality is then utilized to significantly speed up the calculation of scalar products. For a non-orthogonal basis it is necessary to evaluate all scalar products between all basis vectors, which can be done as

\[
\text{MyBasis.scalar\_product\_matrix}().
\]

\texttt{Polynomial} and \texttt{double} versions of the scalar product matrix are then calculated and saved in the member variables \texttt{P\_spm} and \texttt{d\_spm}. For larger bases, the evaluation can be sped up by only calculating the numerical version using \texttt{MyBasis.scalar\_product\_matrix\_num}().

The content of \texttt{P\_spm} and \texttt{d\_spm} can be explored by using the \texttt{<<} operator, but it may also be saved to a file using the \texttt{Col\_basis} member functions \texttt{write\_out\_P\_spm()} and \texttt{write\_out\_d\_spm()} which write out the result in \texttt{ColorFull} and \texttt{Mathematica}, readable format, by default to a file with a default filename in the directory \texttt{ColorResults}. Alternatively the user may supply a filename as an argument.

While the intended usage of \texttt{ColorFull} is to use color bases, \texttt{ColorFull} can also directly define color amplitudes

\[
\text{Col\_amp\_Ca1(“\{1, 3, 2\}(4, 5)“)}: \quad \langle e^{3}\rangle g_{1} q_{2} \text{tr}(e^{4} e^{5})
\]

\[
\text{Col\_amp\_Ca2(“\{1, 3, 4, 5, 2\}“): \quad \langle e^{3}\rangle e^{4} e^{5} e^{5} q_{1} q_{2}
\]

and evaluate scalar products using the \texttt{scalar\_product} member function in the \texttt{Col\_functions} library class

\[
\text{Col\_functions\_Col\_fun;}
\]

\[
\text{Col\_fun.scalar\_product(Ca1, Ca1);} \quad (6.7)
\]

\[
\text{Col\_fun.scalar\_product(Ca1, Ca2);}
\]

…giving the results \texttt{TR*Nc^2*CF^2} and \texttt{TR*Nc*CF}, respectively.

In several contexts, such as parton showers, cancellation of infrared singularities in NLO calculations, and recursive methods for calculating amplitudes [21], it is of interest to know the effect of gluon emission on a color structure. This can be calculated by using the \texttt{Col\_functions} member function \texttt{emit\_gluon}

\[
\text{Col\_fun.emit\_gluon(Ca1, 3, 6);} \quad (6.8)
\]

resulting in

\[
\text{[[1, 3, 6, 2](4, 5)] \quad \text{[[1, 6, 3, 2](4, 5)].} \quad (6.9)
\]

The sign conventions here and elsewhere are such that every gluon inserted after the emitter in the quark-line comes with a plus sign and every gluon inserted before comes with a minus sign. In a basis-decomposed calculation, one would be interested in this result decomposed in the large basis required for one additional gluon. Having a trace basis for this larger vector space, this decomposition can be calculated using

\[
\text{Trace\_basis\_Larger\_basis(nq, nq + 1);} \quad (6.10)
\]

\[
\text{Larger\_basis.new\_vector\_numbers(Cs, emitter);} \quad (6.11)
\]

…where \texttt{Cs} is the \texttt{Col\_str} (basis vector in the smaller trace basis) from which the parton emitter emits a new gluon.

Similarly the effect of gluon exchange on a color structure is of interest. \texttt{ColorFull} offers several functions for dealing with this. Starting with an amplitude, the new amplitude after gluon exchange can be obtained as

\[
\text{Col\_fun.exchange\_gluon(Ca1, 1, 4);} \quad (6.12)
\]

resulting in \texttt{TR[[1, 4, 5, 3, 2]] - TR[[1, 5, 4, 3, 2]]}. \texttt{ColorFull} also has a function for directly calculating the color correlator, i.e., given a color amplitude \texttt{|c|}, the quantity \texttt{|c|T_{i} \cdot T_{j}/|c|} where \texttt{T_{i}} describes the effect of attaching a gluon to parton \texttt{i}. For example

\[
\text{Col\_fun.color\_correlator(Ca1, 1, 2);} \quad (6.13)
\]

…results in the \texttt{Polynomial} \texttt{TR(2)*Nc*CF}.

In some situations, such as soft gluon resummation, it is also useful to calculate the soft anomalous dimension matrix, i.e., to have the result of gluon exchange on any basis vector decomposed into the basis. This can be computed automatically using the \texttt{Col\_basis} member function...
color.gamma. The result is contained in a Poly_matrix. For this we need the full basis since color structures not present at LO will appear at NLO, etc. Coding

```plaintext
Trace_basis MyFullBasis(1,3);
MyFullBasis.color_gamma(1,4);  (6.13)
```

for gluon exchange between the partons 1 and 4 results in the matrix

\[
\begin{align*}
\{0, 0, 0, 0, 0, 0, 0, 0, 1 \ast -1 \, \text{TR}, 0, 0\}, \\
\{0, 0, 0, 0, 0, 0, 0, 0, 0 \ast -1 \, \text{TR}\}, \\
\{0, 0, 1 \, \text{TR} \, \text{NC}, 0, 0, 0, 0, 1 \, \text{TR}, 0, 1 \, \text{TR}\}, \\
\{0, 0, 0, 1 \, \text{TR} \, \text{NC}, 0, 0, 1 \, \text{TR}, 0, 0, 1 \, \text{TR}\}, \\
\{0, 0, 0, 0, 0, 0, 0, 0, 0 \ast -1 \, \text{TR}, 0, 0, 0\}, \\
\{0, 0, 0, 0, 0, 0, 1 \ast -1 \, \text{TR}, 0, 0, 0, 0\}, \\
\{0, 1 \, \text{TR}, 0, 0, 1 \, \text{TR}, 0, 0, 1 \, \text{TR} \, \text{NC}, 0, 0, 0, 0\}, \\
\{1 \ast -1 \, \text{TR}, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 0, 0, 1 \ast -1 \, \text{TR}, 0, 0, 0, 0, 0\}, \\
\{0, 1 \ast -1 \, \text{TR}, 0, 0, 0, 0, 0, 0, 0, 0\}.  (6.14)
\end{align*}
\]

In this way all the soft anomalous dimension matrices needed in [22] can easily be recalculated.

While the number of colors in QCD is three, ColorFull can deal with any \(N_c\), both algebraically and numerically. Numerical evaluation is handled by the Col_functions class, using the (private) member variables NC, TR and CF. Thus also \(T_R\) and \(C_F\) can be changed independently. The reason for keeping \(C_F\) as a parameter technically independent of \(N_c\) is that this allows for keeping the color suppressed part of \(C_F - T_R/N_c\), in a consistent way. This choice has proved useful for accounting for sub-leading \(N_c\) effects in several phenomenological studies [23, 24].

To numerically evaluate a Monomial, Polynomial, Poly_vec or Poly_matrix, the Col_functions member functions double_num are used, for example we may want the \(N_c = 3\) version of a scalar product

```plaintext
Col_fun.double_num( 
    Col_fun.scalar_product(Cal, Cal));  (6.15)
```

giving 8.

For comparison, it is of interest to evaluate (squared) amplitudes in the limit \(N_c \rightarrow \infty\). For taking the leading \(N_c\) limit of any of the polynomial classes Polynomial, Poly_vec or Poly_matrix, the Col_functions member function(s) leading can be used. For example we may take the leading \(N_c\) limit of the matrix in Eq. (6.14),

```plaintext
Col_fun.leading( 
    MyFullBasis.color_gamma(1,4));  (6.16)
```

resulting in the diagonal matrix

\[
\begin{align*}
\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 1 \, \text{TR} \, \text{NC}, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 0, 1 \, \text{TR} \, \text{NC}, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \\
\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}.  (6.17)
\end{align*}
\]

For an expression containing \(C_F\), the color suppressed term \(-T_R/N_c\) can be kept in numerical evaluation by setting full_CF to true, Col_fun.set_full_CF(true).

### 7 Interfacing to Herwig via matchbox

ColorFull can also be interfaced to Herwig++ ([7] via the Matchbox component [20], and can be used to treat the hard interaction, as in for example [25], as well as the parton shower itself [23]. When linked to Herwig++, ColorFull is hooked into the boost linear algebra package, enabling a very efficient treatment of numerical linear algebra. From the next major release of Herwig++, ColorFull will be directly shipped with the Herwig++, code.

### 8 ColorFull code

#### 8.1 Operators

In order to simplify calculations and increase readability, ColorFull defines a standard set of operators, listed in Table 2. This includes the arithmetic operators +, - and * as well as the comparison operators, == and !, and the stream operator <<. The comparison operators, == and !, work by comparing data entries term by term, both for the polynomial classes and for the color carrying classes. Thus, for example, two Polynomials only differing by the order of terms are not considered equal.

#### 8.2 Classes for polynomials

The result of color index contraction can always be written as a sum of terms of the form \(N_c^a T_R^b C_F^c \times \text{constant}\) where we allow for negative integers \(a, b, c\). For the purpose of treating contracted color structures, ColorFull has a minimalistic
Monomial $\sim$ Monomial

Table 2 Standard operators for color structure and polynomial classes. (Operators for both orders of arguments are defined for the arithmetic operators $+$, $-$ and $\times$.)

| Operator | Classes |
|----------|---------|
| $+$ and $-$ | Polynomial $\pm$ (Monomial, Polynomial) |
| | Col_amp $\pm$ (Col_str, Col_amp) |
| $+$ | Polynomial $+$ = (Monomial, Polynomial) |
| | Col_amp $+$ = (Col_str, Col_amp) |
| $\times$ | Monomial $\times$ (int, cnum, double, Monomial) |
| | Polynomial $\times$ (int, cnum, double, Monomial, Polynomial) |
| | Quark_line $\times$ (int, cnum, double, Monomial, Polynomial, Quark_line) |
| | Col_str $\times$ (int, cnum, double, Monomial, Polynomial, Quark_line, Col_str) |
| | Col_amp $\times$ (int, cnum, double, Monomial, Polynomial, Quark_line, Col_str, Col_amp) |
| $\ast$ | Monomial $\ast$ = (int, cnum, double, Monomial) |
| | Polynomial $\ast$ = (int, cnum, double, Monomial, Polynomial) |
| | Col_amp $\ast$ = (Col_str, Col_amp) |
| $\ast$ | Monomial $\ast$ = (int, cnum, double, Monomial) |
| | Polynomial $\ast$ = (int, cnum, double, Monomial, Polynomial) |
| | Polynomials $\ast$ = (Col_str, Col_amp) |
| $\ast$ | Monomial $\ast$ = (int, cnum, double, Monomial) |
| | Polynomial $\ast$ = (int, cnum, double, Monomial, Polynomial) |
| | Polynomials $\ast$ = (Col_str, Col_amp) |
| $=$ | Monomial $=$ Monomial |
| | Polynomial $=$ Polynomial |
| | Poly_vec $=$ Poly_vec |
| | Poly_matr $=$ Poly_matr |
| | Quark_line $=$ Quark_line |
| | Col_str $=$ Col_str |
| | Col_amp $=$ Col_amp |
| $\neq$ | Monomial $\neq$ Monomial |
| | Polynomial $\neq$ Polynomial |
| | Poly_vec $\neq$ Poly_vec |
| | Poly_matr $\neq$ Poly_matr |
| | Quark_line $\neq$ Quark_line |
| | Col_str $\neq$ Col_str |
| | Col_amp $\neq$ Col_amp |
| $<$ | Monomial $<$ Monomial |
| | Polynomial $<$ Polynomial |
| | Poly_vec $<$ Poly_vec |
| | Poly_matr $<$ Poly_matr |
| | Quark_line $<$ Quark_line |
| | Col_str $<$ Col_str |
| | Col_amp $<$ Col_amp |
| $\ll$ | Monomial, Polynomial, Poly_vec, Poly_matr, Quark_line, Col_str, Col_amp, Col_basis |

The simplest class for containing contracted color information is a Monomial

Monomial $\sim$ Nc$^{\text{pow Nc} + \text{pow CF}}$ TR$^{\text{pow TR} \times \text{int_part} x \text{cnum_part}}$. \hspace{1cm} (8.1)

The exponents pow Nc, pow TR and pow CF, as well as int_part are of type int and cnum_part is a cnum, an std::complex $<$ double $>$. These member variables carry the information about the actual monomial.

By default, using the standard constructor, Monomial(), a Monomial is set to 1, by letting cnum_part=1,
int_part=1 and setting all powers to 0. A Monomial with different integer part can be obtained by using the constructor taking an int as argument, and a general Monomial can be constructed using the string constructor (see Table 3 in Appendix A).

8.2.2 Polynomial

A sum of Monomials is contained in the class Polynomial

\[
\text{Poly} \sim \text{Monomial}_0 + \text{Monomial}_1 + \cdots
\]

\[
\text{poly} \quad (8.2)
\]

where, technically, the polynomial information is stored in the public member poly, of type polynomial= std::vector<Monomial>. ColorFull is not designed for manipulating polynomials but Polynomial nevertheless has a minimalistic simplify() function which collects terms with equal power of \( \text{int}_\text{part}=1 \) and setting all powers to 0.\(^6\) It also has a remove_CCF() function for replacing C_F with \( T_R N_c - T_R / N_c \). Details about the Polynomial class can be found in Table 4 in Appendix A.

8.2.3 Poly_vec

For dealing with vectors of Polynomials, ColorFull has a class Poly_vec

\[
\text{Poly_vec} \sim (\text{Polynomial}_0, \text{Polynomial}_1, \ldots)
\]

\[
\text{pv} \quad (8.3)
\]

Poly_vec is a container class for functions acting on vectors of Polynomials whereas the actual information is stored in the poly_vec= typedef std::vector< Polynomial> data member, pv. The member functions of Poly_vec, which include simplify() and remove_CCF(), can be found in Table 5 in Appendix A.

8.2.4 Poly_matr

Finally, there is a class for treating matrices of Polynomials, a Poly_matr,

\[
\text{Poly_matr} \sim \{[\text{Polynomial}_{00}, \text{Polynomial}_{01}, \ldots], \ldots, [\text{Polynomial}_{m0}, \text{Polynomial}_{m1}, \ldots], \ldots\}
\]

\[
\text{pm} \quad (8.4)
\]

which stores the matrix information as a vector of vectors of Polynomials, a poly_matr= typedef std::

\^6\) A default Monomial is thus the neutral element under multiplication, rather than under addition.

vector <Poly_vec>. Again, details can be found in the Appendix A, in Table 6.

8.3 Classes containing the color structure

This section describes the building blocks used by ColorFull to treat the color structure. All classes used to carry the color structure, Quark_line, Col_str, Col_amp, including the basis classes Col_basis — from which Trace_type_basis, Trace_basis, Tree_level_gluon_basis and Orthogonal_basis are derived — have the property that the actual information of the color structure is contained in a type with corresponding name, whereas the class acts as a container for related functions. Thus, for example, the color amplitude information in a Col_amp is contained in a col_amp variable.

8.3.1 Quark_line

The most basic color carrying class is a Quark_line. Loosely speaking, a Quark_line is a quark_line times a Polynomial,

\[
\text{Quark_line} \sim \text{Polynomial} \times \text{quark_line}
\]

\[
\text{Poly} \times \text{ql} \quad (8.5)
\]

where the quark_line (a std::vector<int>) ql contains the actual quark-line information, together with the boolean variable open which is true if the quark-line is open, and false for a trace over gluon indices. The quark_line is multiplied by a Polynomial. For example, we may have the closed Quark_line

\[
(N_c^2-1) (1,2,3,4) \sim (N_c^2-1) \text{tr}(f_1 f_2 f_3 f_4)
\]

\[
(8.6)
\]

or the open Quark_line

\[
N_c T_R \{1,3,4,2\} \sim N_c T_R (f_1 f_4)^g_{\{q_1 q_2\}}.
\]

\[
(8.7)
\]

Quark_lines may be created using the Quark_line(std::string) constructor. Closed quark-lines are denoted by standard parenthesis, whereas curly brackets represent open quark-lines. For example, we may write

\[
\text{Quark_line QL1}("(N_c^2-1) (1,2,3,4"));
\]

\[
\text{Quark_line QL2}("N_c T_R \{1,3,4,2\} "); \quad (8.8)
\]

for the above quark-lines. Among the quark_line member functions we especially note the functions for contracting neighboring and next to neighboring gluon indices, contract_neighboring_gluons() and contract_next_neighboring_gluons(), as well as the normal_order() member function which orders a closed Quark_line such that the smallest gluon index is written...
first. Thus, for example

Quark_line Ql("(2, 3, 4, 5, 1)");
Ql.contract_neighboring_gluons();
Ql.normal_order();

results in \( \text{CF}(1, 2, 3, 5) \). The other public member functions are listed in Table 7 in Appendix A.

8.3.2 Col_str

A general color amplitude consists not only of one quark-line, but of a linear combination of products of Quark_lines. One term in this linear combination, a Polynomial times a product of closed and open quark-lines is contained in a Col_str,

\[
\text{Col}\_\text{str} \sim \text{Polynomial} \times \text{col}\_\text{str}.
\]

Here the actual information about the color structure is carried by a col_str, (technically a vector of Quark_lines), for example we may thus have

\[
(N_c^2 - 1) \{1, 3, 4, 2\}(5, 6)(7, 8)
\]

\[
\sim (N_c^2 - 1) \{t^{53}t^{84}\}_{\text{q}_2} \text{tr}(t^{53}t^{84}) \text{tr}(t^{78}t^{87}).
\]

Like the other color classes, Col_strs may be created using a string constructor

\[
\text{Col}\_\text{str}\text{Cs}"(N_c^2 - 1)
\]

\[
\{[1, 3, 4, 2\}(5, 6)(7, 8)\].
\]

where we note that the col_str is written inside square brackets. The Polynomial should multiply the whole col_str, rather than individual quark_lines.

The Col_str member functions, listed in Table 8 in Appendix A, overlap to a high degree with the Quark_line member functions. In particular there are functions for contracting neighboring and next to neighboring gluons, and a normal_order member function. Apart from normal ordering the individual Quark_lines, the Col_str normal_order() member function sorts the Quark_lines as described in Table 8, where the public members of Col_str are listed.

For Col_str (and Quark_line and Col_amp), there is also a simplify() member function. This function removes quark-lines with 0 quarks, (these are \( N_c \) for closed Quark_lines and 1 for open Quark_lines), normal orders the col_strs, and simplifies the Polynomials. Potential Polynomials multiplying individual Quark_lines are also moved to the Col_str member variable Poly.

8.3.3 Col_amp

The linear combination of Col_strs which in general is needed to keep track of a color structure is contained in a Col_amp. In order to contain scalar results, i.e., terms not containing any color structure (which arise in the process of color contraction) a Col_amp also contains a Polynomial part, Scalar,

\[
\text{Col}\_\text{amp} \sim \text{Polynomial} + \text{col}\_\text{amp}
\]

Scalar \(+ ca\)

where

\[
ca \sim \text{Col}_{0}\_\text{amp} + \text{Col}_{1}\_\text{amp} + \text{Col}_{2}\_\text{amp} + \cdots.
\]

For example, we may have the Col_amp

\[
(1, 2)(3, 4) - N_c\{(-1)(1, 2, 3, 4)
\]

\sim \text{tr}(t^{51}t^{62}) \text{tr}(t^{81}t^{82}) - \frac{1}{N_c} \text{tr}(t^{51}t^{62}t^{81}t^{82}),
\]

(8.13)

where Scalar is 0, and the Polynomials multiplying the Col_strs \((1, 2)(3, 4)\) and \((1, 2, 3, 4)\) are 1 and \(1/N_c\) respectively.

Also the Col_amp class has a string constructor which reads in to the col_amp part according to the syntax

\[
\text{Col}\_\text{amp}\text{Ca}"((1, 2, 3, 4) - 1/N_c[(1, 2)(3, 4)]\).
\]

(8.14)

As for Quark_line and Col_str, Col_amp has member functions for contracting all neighboring and next to neighboring gluons. Gluon contraction between gluons which are further away on the Quark_line may result in a sum of Quark_lines. Such contractions cannot be seen as actions on a single quark_line and are therefore not implemented in the Quark_line and Col_str classes. Instead the Col_amp class contains functions for contracting a gluon or contracting all gluons in the Col_amp. These functions are intended for Col_amps with closed Quark_lines, i.e., quarks should be contracted first, using contract_quarks(Col_amp1, Col_amp2). When calculating a scalar product, all quark indices are thus contracted first, followed by all the gluon indices. The complete list of public member functions is given in Table 9 in Appendix A.

8.4 Col_functions: a library class

Col_functions is a library class containing functions which cannot, in a natural way, be attributed to one class, or functions which act on many classes and are therefore conveniently collected into one class.

In particular, Col_functions contains classes for evaluating scalar products, for numerical evaluation, for taking the leading \( N_c \) limit, and for describing the effect of gluon emission or gluon exchange.
8.4.1 Numerical evaluation

Col_functions is the class which carries numerical values of \( N_c \), \( T_R \) and \( C_F \), stored in the private members \( N_c \), \( TR \) and \( CF \) respectively. To numerically evaluate a Monomial to a double or a complex number (cnum) the function cnum_num or double_num is used.

\[
\text{Col_functions Col_fun;}
\text{Monomial Mon;}
\text{Col_fun.cnum_num(Mon);} \\
\text{Col_fun.double_num(Mon);} \\
\text{\hfill (8.15)}
\]

The syntax for numerical evaluation of Polynomials, Poly_vecs and Poly_mats is identical.

8.4.2 Leading \( N_c \) evaluation

Col_functions also contains functions for taking the leading \( N_c \) limit of the classes Polynomial, Poly_vec and Poly_mats. The leading \( N_c \) terms can be evaluated in two different ways, either by taking the strict \( N_c \to \infty \) limit and dropping all color suppressed terms (default), or by keeping the color suppressed terms in \( C_F = T_R(N_c^2 - 1)/N_c \), while dropping other color suppressed terms. For keeping the full \( C_F \) in numerical evaluations the member variable full_CF must be set to true using set_full_CF(true).

8.4.3 Scalar products

Scalar products are evaluated using the Col_functions member functions scalar_product.

\[
\text{Col_str Cs1("[1, 3, 4, 2](5, 6)\]);}
\text{Col_str Cs2("[1, 3, 4, 5, 6, 2]\)];}
\text{Col_amp Ca1(Cs1);} \\
\text{Col_amp Ca2(Cs2);} \\
\text{Col_fun.scalar_product(Ca1, Ca2);} \\
\text{Col_fun.scalar_product(Cs1, Cs2);} \\
\text{\hfill (8.16)}
\]

in both cases resulting in \( T_R N_c C_F^3 \).

8.4.4 Gluon exchange and gluon emission

For soft gluon resummation, for NLO calculations, and for the cancellation of real emissions and virtual corrections in the soft limit, the color structure associated with gluon exchange plays an important role. The Col_functions class therefore has functions for describing the effect of gluon exchange on a Col_str or on a Col_amp. For example, we may exchange a gluon between parton 3 and 6 in Cs1

\[
\text{Col_fun.exchange_gluon(Cs1, 3, 6);} \\
\text{\hfill (8.17)}
\]

resulting in the Col_amp

\[
-TR\{[1, 5, 6, 3, 4, 2]\} + TR\{[1, 3, 5, 6, 4, 2]\}
+ TR\{[1, 6, 5, 3, 4, 2]\} - TR\{[1, 3, 6, 5, 4, 2]\}.
\]

To understand the signs we note that each time a gluon is inserted before the emitter on a (closed or open) quark-line there is a minus sign, and each time a gluon is inserted after, there is a plus sign. We also note that in this case, the result of gluon exchange on a single color structure gave rise to a linear combination of four color structures, the maximal possible number of color structures from a single col_str [16].

In the context of gluon exchange we also remark that Col_functions can calculate the “color correlator”, \( \langle c|T_i \cdot T_j|c \rangle \) arising when coherently emitting a gluon from parton \( i \) and \( j \) in an amplitude \( |c > \), or when exchanging a gluon between the partons \( i \) and \( j \) in \( |c > \). For example we can calculate the color correlator for exchanging a gluon between parton 1 and 4 in Ca1,

\[
\text{Col_fun.color_correlator(Ca1, 1, 4);} \\
\text{\hfill (8.18)}
\]

resulting in a Polynomial with value \( T_R^2 C_F^2 + T_R^2 N_c C_F^3 \).

Sometimes, for example in the context of a parton shower, one may be interested in the effect of gluon emission itself. Starting in a Col_amp Ca1 and emitting a gluon, 7, from parton 3 this may be found using

\[
\text{Col_fun.emit_gluon(Ca1, 3);} \\
\text{\hfill (8.19)}
\]

giving the Col_amp \( \{[1, 3, 7, 4, 2](5, 6)]\} - \{[1, 7, 3, 4, 2](5, 6)] \).

8.5 Classes for color bases

Although ColorFull may perform calculations with individual Quark_lines, Col_strs and Col_amps, the intended usage is via the color basis classes Trace_basisc (in particular), Tree_level_gluon_basis and Orthogonal_basisc.

As these classes share much of the most important functionality, they all inherit from one base class, Col_basisc. The Trace_basisc and Tree_level_gluon_basis classes inherit from Col_basisc via Trace_type_basisc, whereas Orthogonal_basisc inherits directly from Col_basisc.
8.5.1 Col_basis

The Col_basis class has a col_basis member variable cb for containing the basis vectors.

\[ \text{Col}_\text{basis} \sim \text{col}_\text{basis} \quad \text{cb} \] (8.20)

Col_basis also carries information about the number of quarks and the number of gluons in the public members \( n_q \) and \( n_g \), and (if it has been calculated) the scalar product matrix in polynomial form, \( P_{\text{spm}} \), and in double form, \( d_{\text{spm}} \), as well as the leading (see Sect. 8.4.2) scalar product matrix in polynomial and double versions, leading\( P_{\text{spm}} \) and leading\( d_{\text{spm}} \).

The most important Col_basis member function is probably the scalar_product_matrix() function which calculates the matrix of scalar products between the basis vectors. In its default from, this function uses memoization, as this speeds up the calculations, but this may be circumvented in using the scalar_product_matrix_no_mem() version.

In order not to have to calculate scalar product matrices over and over again Col_basis also has functions for reading in and writing out scalar product matrices, both in numerical and polynomial form.

Col_basis also contains functions for reading in and writing out the basis itself. This is essential for the Orthogonal_basis class which cannot (presently) construct the orthogonal bases, but may read them in.

The decomposition of color amplitudes into bases is done with the (virtual) decompose(Col_amp) member function, which does the decomposition by exploring the coefficients in front of traces and products of traces for Trace_basis and Tree_level_gluon_basis and by evaluating scalar products for the Orthogonal_basis case.

Another important function is the color_gamma function which (using decompose) calculates the soft anomalous dimension matrix, i.e., calculates the matrix describing the effect of gluon exchange between two partons. The result is returned as a Polymatr where component \( i, j \) gives the amplitude for ending up in vector \( i \), if starting in vector \( j \), see also Sect. 6. A list of public members and functions for Col_basis is found in Table 11.

8.5.2 Trace_type_basis

Trace_type_basis is a small helper class for keeping track of functions which are similar for Trace_basis and Tree_level_gluon_basis. It inherits from Col_basis and is inherited from by Trace_basis and Tree_level_gluon_basis. Most importantly, this is where decompose is implemented for these two classes, see Table 12.

8.5.3 Trace_basis

Although the observation that each color amplitude may be decomposed into products of open and closed quark-lines is a guiding principle for ColorFull, the Trace_basis class itself is rather small, containing mainly functions for creating bases.

A trace basis is created by first contracting all \( q\bar{q} \)-pairs in all \( N_q! \) ways, and then attaching gluons to these closed quark-lines, and to additional closed quark-lines, such that at least two gluons are attached to each closed quark-line.

This can be done either directly using a constructor

\[ \text{Trace}_\text{basis \( T_b \)}(2,2); \] (8.21)

or as

\[ \text{Trace}_\text{basis \( T_b \)}; \text{\( T_b \).create_basis}(2,2); \] (8.22)

The result, which can be written out to a user given or default file (write_out_Col_basis) or to cout (using \( << \)) has the basis vectors

\[
\begin{align*}
0 & \{1,5,6,2\}[3,4] \\
1 & \{1,5,6,4\}[3,2] \\
2 & \{1,6,5,2\}[3,4] \\
3 & \{1,6,5,4\}[3,2] \\
4 & \{3,5,6,2\}[1,4] \\
5 & \{3,5,6,4\}[1,2] \\
6 & \{3,6,5,2\}[1,4] \\
7 & \{3,6,5,4\}[1,2] \\
8 & \{1,5,2\}[3,6,4] \\
9 & \{1,5,4\}[3,6,2] \\
10 & \{1,6,2\}[3,5,4] \\
11 & \{1,6,4\}[3,5,2] \\
12 & \{1,2\}[3,4][5,6] \\
13 & \{1,4\}[3,2][5,6] 
\end{align*}
\]

At tree-level, in pure QCD, the last two basis vectors vanish. To create a basis which is only valid up to order \( n_{\text{loop}} \) in pure QCD, we may use the create_basis(n_quark, n_gluon, n_loop) member function. See Table 13 for member functions.

8.5.4 Tree_level_gluon_basis

In the case of gluon-only color structures, charge conjugation implies that each trace must appear with its conjug-
gate, in linear combinations of the form \( \text{tr}[t^g_1 t^g_2 \ldots t^g_n] + (-1)^{N_g} \text{tr}[t^g_n \ldots t^g_2 t^g_1] \). In the trace type basis class `Tree_level_gluon_basis` this is used to reduce the number of basis vectors and speed up calculations. More information can be found in Table 14.

8.5.5 Orthogonal Basis

`ColorFull` can not – in its present form – automatically create multiplet bases. However, orthogonal bases may be read in using the (`Col_basis`) function `read_in_Col_basis(std::string)` member function.

For dealing with orthogonal bases, `ColorFull` offers special functions for calculating the matrix of scalar products and decomposing vectors. The syntax for basis reading is the same as for basis writing. Bases thus appear much as in Eq. (8.23), see Tables 11 and 15.

9 Validation

For a code with order 10,000 lines, validation is essential. For this reason `ColorFull` is continuously validated using a test suite, which aims at testing all the various components. The applied tests starts with checking basic functions for reading in and writing out files, and dealing with polynomials. After this, the creation of bases is tested, and scalar products are tested by changing the order of index contraction, and by switching on and off memoization. The scalar product matrices have further been tested against `ColorMath` [4] and, for processes occurring in the context of [25], also against another private Mathematica code. The functions describing gluon emission and gluon exchange are cross-checked against each other.

10 Conclusion and outlook

`ColorFull`, a C++ stand-alone QCD color algebra package, designed for interfacing with event generators, has been presented. `ColorFull` is based on trace bases, which can automatically be created, and color contraction is performed by repeated usage of the Fierz identity. Employing these bases, one can in principle describe any QCD process. In reality, the scalar product matrices, which may be calculated once and for all, become hard to manage for more than approximately 8 gluons plus \( q\bar{q} \)-pairs. This limitation is inherent for the trace bases, since they are non-orthogonal, and for this reason `ColorFull` is written to be able to load and use orthogonal (multiplet) bases.

`ColorFull` does, however, not – in its present form – perform index contraction in terms of group invariants as described in for example [18,26]. Extending `ColorFull` to inherently construct orthogonal multiplet bases and efficiently perform index contraction using \( 3j \) and \( 6j \) coefficient may speed up the treatment of QCD color space very significantly.

Acknowledgments Simon Plätzer is thanked for interfacing `ColorFull` to Herwig++ and for writing the build system. I also want to thank Simon Plätzer and Johan Grönqvist for many valuable discussions on the organization of this code. This work was supported by a Marie Curie Experienced Researcher fellowship of the MCnet Research Training network, contract MRTN-CT-2006-035606, by the Helmholtz Alliance "Physics at the Terascale" and by the Swedish Research Council, contract number 621-2010-3326, 621-2012-27-44 and 621-2013-4287.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. Funded by SCOAP3.

Appendix A: Class member functions

The (public) members and functions of the `ColorFull` classes are listed below in Tables 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 and 15.
| Data members | Content |
|--------------|---------|
| cnum cnum_part | Complex number multiplying the monomial. |
| int int_part | Integer multiplying the monomial, can be 0. |
| int pow_CF | Power of \( CF=TR*(Nc^2-1)/Nc \). |
| int pow_Nc | Power of the number of colors. |
| int pow_TR | Power of TR in the Monomial. |

| Constructors | Effect |
|--------------|--------|
| Monomial() | Default constructor that sets \( \text{int\_part}=\text{cnum\_part}=1 \), and \( \text{pow\_Nc}=\text{pow\_TR}=\text{pow\_CF}=0 \). |
| Monomial(double dnum) | Constructor using a double. The \( \text{cnum\_part} \) member is set to contain the value. |
| Monomial(int num) | Constructor using an int. The \( \text{int\_part} \) member is set to contain the value. |
| Monomial(std::string str) | Constructor taking a string as argument. The argument should be of the form in for example \( "-(20*TR^5)/Nc" \) or \( "-20 \ TR^5/Nc" \) or \( "20 / TR^(-5)Nc^(1) CF^(3)" \). \textit{Note:} All spaces and all \( * \) are ignored, except in \( *(-1) \) and \( *-1 \), which are understood as \( *(-1) \). Everything standing after \( / \) is divided out, whereas everything standing before \( / \) is multiplied with. Parentheses are ignored unless they appear in powers, directly after \( ^ \). No spaces are allowed inside the powers. If the string contains no information or is empty, the \textit{Monomial} is set to 1, i.e., \( \text{pow\_TR} = \text{pow\_Nc} = \text{pow\_CF} = 0 \), \( \text{int\_part} = 1 \), \( \text{cnum\_part} = 1.0 \). (Expanded Mathematica expressions are of this form.) |

| Member functions | Effect |
|------------------|--------|
| void conjugate() | Take the complex conjugate. Note that this changes the \textit{Monomial} itself. |
| void read_in_Monomial(std::string filename) | Function for reading in the \textit{Monomial} from the file \textit{filename}. The syntax is as for \textit{Monomial(std::string)}, and comments starting with \# are allowed at the top of the file. |
| void write_out_Monomial(std::string filename) const | Function for writing out the \textit{Monomial} to a file with name \textit{filename}. |
### Table 4  Public data members and functions of the class Polynomial

| Data member                                                                 | Content                                                                                                                                                                                                 |
|-----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| polynomial poly where polynomial is typedef std::vector<Monomial>          | Contains the polynomial, a sum of Monomials. An empty polynomial is defined as 1, to get 0, multiply with 0.                                                                                           |

| Constructors                                                                 | Effect                                                                                                                                                                                                 |
|-----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Polynomial()                                                                | Default constructor, leaves polynomial empty (=1).                                                                                                                                                     |
| Polynomial(const std::string str)                                           | Constructor allowing setting the Polynomial by using a string, should be used as for example: "Polynomial Poly("(-20*TR^(5))/Nc + 28*Nc*TR^(5) - 10*Nc^3*TR^(5)\)". The Monomials should be separated by + or -, see also the Monomial string constructor. |
| Polynomial(double dnum)                                                     | Constructor allowing setting the Polynomial using a double. The Polynomial gets one Monomial where the real part of cnum_part equals dnum.                                                            |
| Polynomial(int num)                                                         | Constructor allowing setting the Polynomial using an int. The Polynomial gets one Monomial where int_part has the value num.                                                                            |

| Member functions                                                            | Effect                                                                                                                                                                                                 |
|-----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void append(const Monomial Mon)                                             | Adding a Monomial term.                                                                                                                                                                               |
| const Monomial& at(int i) const                                             | Returns the Monomial at place i.                                                                                                                                                                      |
| Monomial& at(int i)                                                         | Returns the Monomial at place i.                                                                                                                                                                      |
| void clear()                                                                | Erases the information in polynomial.                                                                                                                                                                 |
| void conjugate()                                                            | Takes the complex conjugate of the polynomial. Note that this changes the Polynomial itself.                                                                                                         |
| bool empty()                                                                | Is the polynomial empty?                                                                                                                                                                              |
| void erase(int i)                                                           | Erase the Monomial at place i.                                                                                                                                                                        |
| void normal_order()                                                         | Orders terms in Polynomial in a unique form, first according to pow_Nc+pow_CF, then according to pow_Nc (for same pow_Nc+pow_CF) then according to int_part*pow_Nc+pow_CF then according to int_part, and finally according to pow_TR. |
| void read_in_Polynomial(std::string filename)                              | Function for reading in the Polynomial from the file filename. Comments starting with # are allowed at the top of the file.                                                                          |
| void remove_CF()                                                            | Replaces CF with TR*Nc-TR/Nc.                                                                                                                                                                         |
| void simplify()                                                             | Collects terms with the same power of TR, Nc and CF.                                                                                                                                                 |
| int size()                                                                  | Returns the number of terms in the Polynomial.                                                                                                                                                         |
| void write_out_Polynomial(std::string filename) const                      | Function for writing out the Polynomial to a file with name filename.                                                                                                                                  |
### Table 5  Public members and functions in the class Poly_vec

| Data member                      | Content                                                                 |
|----------------------------------|-------------------------------------------------------------------------|
| poly_vec pv                      | To actually contain the vector of Polynomials.                           |
| where poly_vec is:               |                                                                         |
| typedef std::vector<Polynomial>   |                                                                         |

| Constructors                      | Effect                                                                 |
|-----------------------------------|-------------------------------------------------------------------------|
| Poly_vec()                        | Default constructor, leaves pv empty.                                  |
| Poly_vec(poly_vec poly_v)         | Makes a Poly_vec of a poly_vec.                                        |

| Member functions                  | Effect                                                                 |
|-----------------------------------|-------------------------------------------------------------------------|
| void append(Polynomial Poly)      | Appends a Polynomial to data member pv.                                |
| const Polynomial& at(int i) const | Returns the Polynomial at place i.                                     |
| Polynomial& at(int i)             | Returns the Polynomial at place i.                                     |
| void clear()                      | Erases the information in the vector.                                  |
| void conjugate()                  | Conjugates the Poly_vec.                                               |
| bool empty() const                | Is the vector empty?                                                   |
| void normal_order()               | Normal orders all Polynomials in the poly_vec member pv (uses the Polynomial::normal_order function). |
| void read_in_Poly_vec(std::string filename) | Reads in a Polynomial vector of the form \{Poly1, Poly2, \ldots\} to the member pv from the file filename. Comments starting with \# are allowed at the top of the file. |
| void remove_CF()                  | Remove CF in the poly_vec member pv, i.e., replace CF by TR*Nc-TR/Nc.  |
| void simplify()                   | Simplifies all polynomials in the poly_vec member pv (uses the simplify member function in Polynomial). |
| uint size() const                 | Returns the number of Polynomials in the vector, i.e., the size of the member pv. |
| void write_out_Poly_vec(std::string filename) const | Writes out the vector to the file filename. |
Table 6  Public members and functions of the class Poly_matr

| Data member | Content |
|-------------|---------|
| poly_matr pm where poly_matr is: typedef std::vector<Poly_vec>| To actually contain the matrix of Polynomials. |

| Constructor | Effect |
|-------------|---------|
| Poly_matr() | Default constructor, leaves pm empty. |

| Member functions | Effect |
|------------------|--------|
| Poly_vec& at(int i) | Returns the Poly_vec at place i. |
| const Poly_vec& at(int i) | Returns the Poly_vec at place i. |
| Polynomial& at(int i, int j) | Returns the matrix element at i, j. |
| const Polynomial& at(int i, int j) | Returns the matrix element at i, j. |
| void append(Poly_vec Pv) | Appends a Poly_vec to the data member pm. |
| void clear() | Erases the matrix information. |
| void conjugate() | Conjugates the matrix. |
| bool empty() | Is the matrix, stored in pm, empty? |
| void normal_order() | Normal orders all quark_lines in the poly_matr member pm (uses Polynomial::normal_order). |

void read_in_Poly_matr(std::string filename)  
Reads in the matrix from the file filename. The file should be of the format 
\{Poly11,...,Poly1n\},  
...,  
\{Polyn1,...,Polynn\},  
and may contain comment lines starting with # at the top.

void remove_CF()  
Removes CF in the poly_matr member pm, i.e., replaces CF by TR*Nc-TR/Nc.

void simplify()  
Simplifies all quark_lines in the poly_matr member pm, (uses Polynomial.simplify).

uint size()  
Returns the size of the matrix, the number of Poly_vecs in the member pm.

void write_out_Poly_matr( std::string filename) const  
Writes out the matrix to the file filename.
Table 7  Public members and functions of the class Quark\_line

| Data members                      | Content                                                                                                                                 |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| quark\_line ql where quark\_line is: typedef std::vector<int> | To actually contain the color information, in order \( \{q, g_1, g_2, \ldots, g_n, \bar{q}\} \) or \( \{g_1, g_2, \ldots, g_n\} \). |
| bool open                         | Is the string open, with a quark in the beginning and an antiquark in the end, or not?                                                |
| Polynomial Poly                   | Polynomial factor, multiplying the \texttt{quark\_line}.                                                                                                                                 |

| Constructors                      | Effect                                                                                                                                 |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Quark\_line( )                    | Default constructor, leaves \texttt{ql} empty.                                                                                           |
| Quark\_line(const std::string)    | Constructor used to set the color structure using a string. The string should be of form \texttt{Polynomial*quark\_line}, used as \texttt{Quark\_line Ql("5*TR*Nc^2 \{1,6,7,2\}");} for an open \texttt{Quark\_line} with a quark with number 1, two gluons with number 6 and 7, and a \( \bar{q} \) with number 2. For a closed \texttt{Quark\_line} the syntax is \texttt{Quark\_line Ql("1,2,3")}; The integers should be positive. The \texttt{Polynomial} should be in a format which it is readable by the \texttt{Polynomial(std::string)} constructor. |

| Member functions                  | Effect                                                                                                                                 |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Quark\_line after(int j) const    | Returns a \texttt{Quark\_line} where the \texttt{ql} member is changed to contain only partons after place \( j \).                      |
| void append(int p)                | Appends parton \( p \) to the \texttt{Quark\_line}.                                                                                     |
| void append(std::vector<int> in\_ql) | Appends a whole \texttt{quark\_line} to the \texttt{Quark\_line}.                                                                       |
| int at(int j) const               | Returns the parton at place \( j \). For closed \texttt{quark\_lines} \( j \) may be between \( -\text{size} \) and \( 2*\text{size} \). |
| Quark\_line before(int j) const   | Returns a \texttt{Quark\_line} where the \texttt{ql} member is changed to contain only partons before place \( j \).                    |
| void clear()                      | Erase the information in \texttt{quark\_line ql}.                                                                                         |
| void conjugate()                  | Conjugates the \texttt{Quark\_line} by reversing the \texttt{quark\_line} \texttt{ql} and conjugating the \texttt{Polynomial Poly}.       |
| void contract\_neighboring\_gluons(int j) | Contracts neighboring gluons in the \texttt{Quark\_line} starting at \( j \), only intended for closed \texttt{Quark\_lines}.          |
| void contract\_neighboring\_gluons() | Function for contracting neighboring gluons in a \texttt{Quark\_line} starting at place 0, and looking everywhere, only intended for closed \texttt{Quark\_lines}. |
Table 7 continued

| Function Name                                      | Description                                                                 |
|---------------------------------------------------|-----------------------------------------------------------------------------|
| `void contract_next_neighboring_gluons(int j)`    | Contracts next to neighboring gluons in the **Quark_line**, starting at place j (i.e. checking first gluon j and j+2). Also looks for new neighbors, only intended for closed **Quark_lines**. |
| `void contract_next_neighboring_gluons()`         | Contracts neighboring and next to neighboring gluons in the **Quark_line**, starting with contracting neighbors, only intended for closed **Quark_lines**. |
| `bool empty() const`                              | Is the **quark_line** empty?                                                |
| `void erase(int i)`                                | To erase the parton at place i.                                             |
| `void insert(int j, int p)`                        | Inserting parton p at place j.                                             |
| `void normal_order()`                              | Orders a closed **quark_line**, such that the smallest gluon index stands first (i.e., uses that the trace is cyclic). |
| `void prepend(int p)`                              | Prepends parton p to the **Quark_line**.                                   |
| `void prepend(std::vector<int> in_ql)`             | Prepends a whole **quark_line** to the **Quark_line**.                      |
| `void read_in_Quark_line(std::string filename)`   | Function for reading in the **Quark_line** from the file `filename`. The syntax for the **Quark_line** is the same as for the **Quark_line** string constructor. |
| `uint size() const`                                | The size of the **quark_line**.                                             |
| `int smallest(const Quark_line& Q1, const Quark_line& Q2) const` | Function for finding the "smallest" **Quark_line** of Q1 and Q2, used for deciding which **Quark_line** should stand first when normal ordering Col_strs. This function does **not** first normal order the **Quark_lines**. If only one is open, that **Quark_line** should stand first. If both are open or both are closed, the longest **Quark_line** should stand first. If the size is the same, the **Quark_line** with smallest starting number should stand first. If the first number is the same, check the second number, then the third etc. 1 is returned if Q1 should stand first, and 2 if Q2 should stand first. If Q1==Q2, 0 is returned. |
| `std::pair<Quark_line, Quark_line> split_Quark_line(int j1, int j2) const` | Function for splitting a closed **Quark_line** into two **Quark_lines**. The gluons at j1 and j2 are removed in the split. May create 1-rings and 0-rings. |
| `void write_out_Quark_line(std::string filename) const` | Function for writing out the **Quark_line** to a file with name `filename`. |
Table 8  Public data members an functions of the class Col_str

| Data members | Content |
|--------------|---------|
| col_str cs  | For containing the information about the color structure, a product of Quark_lines, contained in a vector of quark_lines. |
| typedef std::vector<Quark_line> | |
| Polynomial Poly | Polynomial factor multiplying the whole product of Quark_lines. |

| Constructors | Effect |
|-------------|--------|
| Col_str() | Default constructor, leaves cs empty. |
| Col_str(Quark_line Ql) | Make a Col_str of a Quark_line. |
| Col_str(const std::string str) | Constructor for setting the color structure using a string. Should be used as: Cs("Nc*TR" (3) [{1,2,3,4}(5,6)(7,8)]), i.e., the argument should be Polynomial * col_str. (The Polynomial should multiply the whole col_str, i.e., stand outside the []-brackets). |

| Member functions | Effect |
|------------------|--------|
| const Quark_line& at(int i) | Returns the Quark_line at place i. |
| Quark_line& at(int i) | Returns the Quark_line at place i. |
| int at(int i, int j) const | Returns the parton at place j in in Quark_line i. |
| void append(Quark_line Ql) | Appends a Quark_line to the data member cs. |
| void append(col_str cs_in) | Append the content of a col_str to the cs of the Col_str. |
| void clear() | Erase information in col_str. |
| void conjugate() | Function for conjugating the Col_str by conjugating each Quark_line in cs, as well as the Polynomial Poly. |
| void contract_2_rings() | Function for contracting gluon indices in closed Quark_lines with only 2 gluons. This removes the 2-ring, replaces one of the gluon indices, and multiplies with a factor tr[τ^a τ^a] = TR (no sum), only intended for fully contractable Col_strs. |
| void contract_next_neighboring_gluons() | Contracts neighboring and next to neighboring gluons in each Quark_line in the Col_str, starting with contracting neighbors. This function should only be used on Col_strs with only closed Quark_lines. |
| void contract_quarks(const Col_str Cs1, const Col_str Cs2) | Function for contracting quarks between two color structures Cs1 and Cs2. The result is stored in the Col_str itself. |
| bool empty() const | Is the col_str empty? |
Table 8 continued

| Method | Description |
|--------|-------------|
| `void erase(int i)` | Erases the `Quark_line` at place i. |
| `void erase(int i, int j)` | Erases the parton at position j in `Quark_line` i. |
| `void erase(std::pair<int, int> place)` | Erases the parton located at place. |
| `std::string find_kind(int p) const` | Finds out if a parton is a quark, anti-quark or gluon, returns "q", "qbar" or "g" respectively. This function does not loop over all partons, but assumes that the parton is a gluon if it is in a closed `Quark_line`, or if the `Quark_line` is open but the parton cannot be found in the ends. |
| `std::pair<int, int> find_parton(int part_num) const` | Locates the parton with number `part_num` in a `col_str`. |
| `bool gluons_only() const` | Checks if the amplitude only has gluons, i.e., if all `Quark_lines` are closed. |
| `void insert(int i, int j, int part_num)` | To insert the parton `part_num` in `quark_line` i at place j. |
| `bool left_neighbor(int p1, int p2) const` | Function for telling if parton `p2` stands to the left of parton `p1`. |
| `int longest_quark_line() const` | Returns the length of the longest `Quark_line` in the `col_str`. |
| `int n_gluon() const` | Counts the number of gluons in a `Col_str`. Counts all gluon indices, both free and contractable. |
| `int n_quark() const` | Counts the number of quarks (=number of anti-quarks) in a `Col_str`. Counts all quark indices, both free and contracted. |
| `bool neighbor(int p1, int p2) const` | Function for telling if the partons `p1` and `p2` are neighbors. |
| `void normal_order()` | Normal orders the `Col_str` by first normal ordering individual `Quark_lines` and then normal ordering different `Quark_lines` in `cs`. For the ordering, see the member function `smallest` in this class and in the `Quark_line` class. |
| `void read_in_Col_str(std::string filename)` | Function for reading in the `Col_str` from the file `filename`. |
| `void remove_0_rings()` | Removes `Quark_lines` without partons, equal to Nc (closed) or 1 (open). |
| `void remove_1_rings()` | Removes `Quark_lines` with only one gluon as tr(t^a) = 0. |
| `void replace(int old_ind, int new_ind)` | Replaces the parton index `old_ind` with `new_ind`. |
| `bool right_neighbor(int p1, int p2) const` | Function for telling if parton `p2` stands to the right of parton `p1`. |
| Function | Description |
|----------|-------------|
| `void simplify()` | Removes 0- and 1-rings, moves factors multiplying the individual `Quark_lines` to multiply the `col_str` instead (i.e., being stored in `Poly`), simplifies the `Polynomial` and normal orders the `quark_lines`. |
| `uint size()` | The size of the `col_str`, number of `quark_lines`. |
| `int smallest(const Col_str& Cs1, const Col_str& Cs2) const` | Finds out the “smallest” `Col_str` of two `Col_strs`, i.e., which `Col_str` should stand first in a normal ordered `Col_amp` or basis. Returns 1, if `Cs1` should stand before `Cs2` and 2 if `Cs2` should stand before `Cs1`. Both `Col_strs` have to be normal ordered for the result to be unique. The `Col_strs` are ordered by (1) number of `Quark_lines` (2) if the `Quark_line` at place 0,1,2... is open or not (3) the size of the `Quark_line` at place 1,2,3... (4) the parton numbers in the `Quark_lines` at place 1,2,3..., i.e., first the first parton in the first `Quark_line` is checked and last the last parton in the last `Quark_line`. The function returns 0 if `Cs1==Cs2`. |
| `void write_out_Col_str(string filename) const` | Function for writing out the `Col_str` to a file with name `filename`. |
Table 9  Public members and functions of the class Col_amp

| Data members | Content |
|--------------|---------|
| `col_amp ca` | To actually contain the information about the Col_strs, ca=Cs1+Cs2+Cs3+.... |
| where `col_amp` is: `typedef std::vector<Col_str>` | | |

| Polynomial Scalar | Scalar is a Polynomial for collecting color factors appearing when the color structure has been fully contracted. The full color amplitude is Scalar+Cs1+Cs2+Cs3.... Scalar should thus be non-zero only if all indices can be contracted. | |

| Constructors | Effect |
|--------------|---------|
| `Col_amp()` | Default constructor, sets Scalar = 0, and leaves ca empty. |
| `Col_amp(Col_str Cs)` | Constructor converting a Col_str to a Col_amp. |
| `Col_amp(const std::string str)` | Constructor taking a string as argument. The string should be of the form Polynomial1*col_str1+Polynomial2*col_str2, for example: Ca("Nc*[(1,3,4,2)]+1/Nc [(1,4)(3,2)]"). (The Polynomials should multiply the whole col_strs, and thus stand outside the []-brackets.) |

| Member functions | Effect |
|------------------|--------|
| `void append(col_amp ca_in)` | Appends the Col_strs in ca_in to the col_amp member ca. |
| `const Col_str & at(int i)` | Returns the Col_str at place i. |
| `Col_str & at(int i)` | Returns the Col_str at place i. |
| `void clear()` | Erases the information about the color amplitude, stored in ca. |
| `void collect_col_strs()` | Compares col_strs in the Col_amp to collect similar col_strs and only store once in ca. |
| `void conjugate()` | Function for taking the conjugate of the Col_amp by conjugating each Col_str in ca and the Polynomial member Scalar. |
| `void contract_2_rings()` | Contract closed Quark_lines with only 2 gluons in each Quark_line in each Col_str in the Col_amp. This removes the 2-ring, replaces one of the gluon indices and multiplies with a factor tr[t^a_t^a] = TR (no sum), only intended for fully contractable Col_amps. |
| `void contract_a_gluon()` | Contracts one gluon, the first gluon in the first Quark_line (in each Col_str), only intended for closed Quark_lines. |
Table 9 continued

| Function | Description |
|----------|-------------|
| `void contract_all_gluons()` | Function for contracting all gluon indices in a `Col_amp`, only intended for closed `Quark_lines`. |
| `void contract_next_neighboring_gluons()` | Contracts up to next to neighboring gluons in each `Quark_line` in each `Col_str` in each `Col_amp`, only intended for closed `Quark_lines`. |
| `void contract_quarks(const Col_amp Ca1, const Col_amp Ca2)` | Function for contracting the (anti-)quarks in Ca1 with those in Ca2. The results is saved in the `Col_amp` itself. |
| `void contract_Quark_line_gluons()` | Function for contracting gluon indices within the `Quark_lines`. Checks only for one pair in each `Quark_line`. |
| `bool empty() const` | Is the `col_amp` ca empty? |
| `void erase(int i)` | Erases the `Col_str` at place i. |
| `bool gluons_only() const` | Checks if the `Col_amp` only contains gluons, i.e., if all `Quark_lines` are closed. |
| `int longest_quark_line() const` | Returns the length of the longest `Quark_line` in any `Col_str`. |
| `int n_gluon() const` | Returns the number of gluons in the `Col_amp` as the number of gluons in the first `Col_str`. Note that the other `Col_strs` could have a different number of (contracted) gluons. (Intended for tree-level `Col_amps` with only one `Col_str`.) |
| `int n_gluon_check() const` | Returns the number of gluons in the `Col_amp` after checking that each `Col_str` has the same number of gluons. |
| `int n_quark() const` | Returns the number of quarks in the `Col_amp` as the number of quarks in the first `Col_str`. Note that the other `Col_strs` could have a different number of (contracted) quarks. (Intended for tree-level `Col_amps` with only one `Col_str`.) |
| `int n_quark_check() const` | Returns the number of quarks in the `Col_amp` after checking that each `Col_str` has the same number of quarks. |
| `void normal_order()` | Normal orders the individual `col_strs` and then orders the `Col_strs` using the order defined in the `Col_str` member function `smallest`. |
| `void normal_order_col_strs()` | Normal orders all `col_strs` in `ca`. |
| `void read_in_Col_amp(std::string filename)` | Reads in the `Col_amp` to the member `ca` from the file `filename`. (This is intended for reading in an actual color amplitude, nothing is read in to the `Polynomial` member `Scalar`.) Comments starting with `#` are allowed at the top of the file. |
| `void remove_0_rings()` | Removes `quark_lines` with no gluons, these are `Nc` if closed, and defined to be 1 if open. |
Table 9 continued

| Function                          | Description                                                                 |
|----------------------------------|-----------------------------------------------------------------------------|
| void remove_1_rings()            | Removes Col_strs with quark_lines with only 1 gluon, these are 0 as \( \text{tr}[t^a] = 0 \). |
| void remove_empty_Col_strs()     | Removes empty Col_strs. An empty Col_str means that all indices have been contracted, so the Col_str is equal to its Polynomial, which is moved to the Scalar part of the Col_amp. |
| void simplify()                  | Function for simplifying an amplitude. Removes 0 and 1-rings, compares col_strs, removes Col_strs multiplying 0, and simplifies Polynomials of the individual Col_strs. |
| uint size() const                | The size of the col_amp ca.                                                  |
| void write_out_Col_amp(std::string filename) const | Function for writing out the Col_amp to a file with name filename. |
Table 10  Some (private) data members and the public functions of the library class Col\_functions

| Data members (private) | Content |
|------------------------|---------|
| double CF              | The value of $C_F = T_R(N_c^2 - 1)/N_c$, changed by the set\_CF function. Note that CF can be changed independently of Nc. |
| bool full\_CF           | While finding the leading terms in a Polynomial one may want to keep the full value of CF, $TR\cdot N_c - TR/N_c$, or only keep the leading Nc term $TR\cdot N_c$ (default). The switch full\_CF is used by the Polynomial version of leading (and hence also by the Poly\_vec and Poly\_matr versions etc.). The leading functions replaces CF by $TR\cdot N_c$ if full\_CF is false (default) while evaluating the leading terms. If full\_CF is true, CF is replaced by $TR\cdot N_c - TR/N_c$. Clearly this affects the result of subsequent numerical evaluation. In the Col\_basis class (and derived) the matrix version of leading is used to evaluate scalar product matrices. |
| double Nc              | The number of colors, used in numerical results, changed by the set\_Nc function. |
| double TR              | The trace convention $\text{tr}(t^a t^a) = TR$ (no sum), the normalization of the SU($N_c$) generators, to be used in numerical evaluation. This value can be changed by the set\_TR function. |

| Constructor | Effect |
|-------------|--------|
| Col\_functions() | Default constructor, sets Nc=3, TR=0.5, CF=4.0/3.0 and full\_CF=false. |

| Member functions | Effect |
|------------------|--------|
| cnum cnum\_num(const Monomial& Mon) const | Numerically evaluates a Monomial using the Nc, TR and CF variables. |
| cnum cnum\_num(const Polynomial& Poly) const | Numerically evaluates a Polynomial, using the Nc TR and CF variables. |
| cvec cnum\_num(const Poly\_vec& Pv) const | Numerically evaluates a Poly\_vec (vector of Polynomials), using cnum\_num(Polynomial). |
| cmatr cnum\_num(const Poly\_matr& Pm) const | Numerically evaluates a Poly\_matr (vector of Poly\_vec), using cnum\_num(Poly\_vec) for each Poly\_vec. |
Table 10  continued

| Function Name                                                                 | Description                                                                 |
|-------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| Polynomial color_correlator(const Col_amp Ca, int i, int j) const             | Calculates $\langle c|T_i T_j|c\rangle$, the "color correlator" relevant for coherent gluon emission from parton $i$ and parton $j$, or gluon exchange between $i$ and $j$. The $Ca$ thus corresponds to $|c\rangle$, and $i$ and $j$ are the partons involved in the emission (exchange). |
| double double_num(const Monomial & Mon) const                                 | Numerically evaluates a Monomial to a double, using the data members $Nc$, $CF$ and $TR$. |
| double double_num(const Polynomial & Poly) const                              | Numerically evaluates a Polynomial to a double, using the data members $Nc$, $CF$ and $TR$. |
| dvec double_num(const Poly_vec & Pv) const                                     | Numerically evaluates a Poly_vec (vector of Polynomial).                      |
| dmatr double_num(const Poly_matr & Pm) const                                   | Numerically evaluates a Poly_matr (vector of Poly_vec).                       |
| Col_amp emit_gluon(const Col_str & in_Col_str, int emitter, int g_new) const  | Function for emitting a gluon from a Col_str. When the gluon is inserted before the emitter in a Quark_line, the amplitude comes with a minus sign. |
| Col_amp emit_gluon(const Col_amp & Ca_in, int emitter, int g_new) const        | Function for emitting a gluon from a Col_amp. When the gluon is inserted before the emitter in a Quark_line, the amplitude comes with a minus sign. |
| Col_amp exchange_gluon(const Col_str & Cs, int p1, int p2) const               | Function for exchanging a gluon between the partons $p1$ and $p2$ in the Col_str Cs. When the gluon is inserted before the emitter in a Quark_line, the amplitude comes with a minus sign. |
| Col_amp exchange_gluon(const Col_amp & Ca, int p1, int p2) const               | Function for exchanging a gluon between two partons $p1$ and $p2$ in the Col_amp Ca. When the gluon is inserted before the emitter in a Quark_line, the amplitude comes with a minus sign. |
| double get_CF() const                                                         | Returns the value of $CF$.                                                   |
| double get_Nc() const                                                          | Returns the number of colors.                                                |
| double get_TR() const                                                          | Returns the normalization of the generators, $\text{tr}(t^a t^b) = TR \delta^{ab}$. |
| bool get_full_CF() const                                                       | Returns $true$ if full_CF is true and $false$ otherwise.                    |
| int factorial(int i) const                                                    | The factorial of an int, $0!$ is defined as $1$.                            |
| **Table 10 continued** | **Description** |
|------------------------|-----------------|
| `Polynomial leading(const Polynomial & Poly) const` | Takes the leading `Nc` terms of a `Polynomial`, i.e., keeps the `Monomials` with the highest power of `Nc` plus `CF`. If `full_CF` is `false` (default), `CF` is replaced by `TR*Nc`. If `full_CF` is `true`, `CF` is replaced by `TR*Nc-TR/Nc`. |
| `Poly_vec leading(const Poly_vec& Pv) const` | Takes the leading part of a `Poly_vec`. Keeps only `Monomials` with maximal power of `CF` plus `Nc`, uses `leading(const Polynomial & Poly)`. If `full_CF` is `false` (default), `CF` is replaced by `TR*Nc`. If `full_CF` is `true`, `CF` is replaced by `TR*Nc-TR/Nc`. Note that taking the leading terms of a `Poly_vec` is not necessarily the same as taking the leading terms of each `Polynomial`. |
| `Poly_matr leading(const Poly_matr& Pm) const` | Takes the leading part of a matrix of `Polynomials`, keeping only those with maximal power of `CF` plus `Nc`. If `full_CF` is `false` (default), `CF` is replaced by `TR*Nc`. If `full_CF` is `true`, `CF` is replaced by `TR*Nc-TR/Nc`. Note that taking the leading terms of a `Poly_matr` is not necessarily the same as taking the leading terms in each `Poly_vec`. |
| `int leading_Nc_pow(const Polynomial & Poly) const` | Function for finding the leading power of `Nc` in a `Poly_vec`, i.e., the power of `Nc` plus the power of `CF`. |
| `int leading_Nc_pow(const Poly_vec & Pv) const` | Function for finding the leading power of `Nc` in a `Poly_vec`. |
| `Polynomial Polynomial_cnum_num(const Polynomial & Poly) const` | Numerically evaluates a `Polynomial` using the value of the data member `Nc`, and stores in the format of a `Polynomial` with only one term with only a numerical part. |
| `Poly_vec Poly_vec_cnum_num(const Poly_vec & Pv) const` | Numerically evaluates a `Poly_vec` (vector of `Polynomial`) and stores in the form of a `Poly_vec`, uses `polynomial_cnum_num(Pv.at(p))` for each `Polynomial`. |
| `Poly_matr Poly_matr_cnum_num(const Poly_matr & Pm) const` | Numerically evaluates a `Poly_matr` (vector of `Poly_vec`) and stores in the form of a `Poly_matr`. |
Table 10  continued

| Function | Description |
|----------|-------------|
| `dvec read_in_dvec(std::string filename) const` | Reads in a numerical vector and saves it as a double matrix, `dmatr`. The file should be of the format `{d1,...,dn}`, and may contain comment lines starting with `#` at the top. |
| `dmatr read_in_dmatr(std::string filename) const` | Reads in a numerical matrix and save it as a double matrix, `dmatr`. The file should be of the format `{d11,...,d1n},
....,
{dn1,...,dnn}` and may contain comment lines starting with `#` at the top. |
| `Polynomial scalar_product(const Col_amp & Ca1, const Col_amp & Ca2) const` | Function for calculating the scalar product between two `Col_amps`. |
| `Polynomial scalar_product(const Col_str & Cs1, const Col_str & Cs2) const` | Function for calculating the scalar product between two `Col_strs`. |
| `void set_Nc(double n)` | Set the number of colors. The value of `CF` is adjusted accordingly. |
| `void set_TR(double tr)` | Sets the normalization of the generators. The value of `CF` is adjusted accordingly. |
| `void set_CF(double cf)` | Sets the value of `CF`. The value of `Nc` is not adjusted accordingly. |
| `void set_full_CF(bool is_full)` | Switch on/off `full_CF`. |
| `void write_out_dmatr(const dmatr & matr, std::string filename) const` | Writes out the double version of a (scalar product) matrix to the file `filename`. |
| `void write_out_dvec(const dvec & dv, std::string filename) const` | Function for writing out a numerical vector to the file `filename`. |
Table 11  Public members and functions of the class Col\_basis

| Data members       | Content                                                                 |
|-------------------|-------------------------------------------------------------------------|
| col\_basis cb     | Contains the information about the basis vectors cb= vector1, vector2.... |
| where col\_basis  |                                                                         |
| is:               |                                                                         |
| typedef std::vector<Col\_amp> |                                                              |
| Col\_functions Col\_fun | Contains the set of Col\_functions used.                          |
| dmatr d\_spm      | To contain the double version of the scalar product matrix.          |
| dmatr leading\_d\_spm | To contain the double version of the leading part of the scalar product matrix. |
| Poly\_matr leading\_P\_spm | To contain the Polynomial version of the leading part of the scalar product matrix. |
| int ng            | The number of gluons, initially set to 0 and (possibly) changed with create\_basis, or while reading in the basis. |
| int nq            | The number of q\_q-pairs, initially set to 0 and (possibly) changed with create\_basis, or while reading in the basis. |
| Poly\_matr P\_spm | To contain the Polynomial version of the scalar product matrix.      |

| Constructor and destructor | Effect                                                                 |
|---------------------------|-------------------------------------------------------------------------|
| Col\_basis()              | Default constructor, sets nq=ng=0, and the private data members trace\_basis = tree\_level\_gluon\_basis = orthogonal\_basis = false. |
| virtual \~Col\_basis()    | Destructor.                                                            |

| Member functions          | Effect                                                                 |
|---------------------------|-------------------------------------------------------------------------|
| Col\_amp& at(const int i) | Returns the Col\_amp (i.e. basis vector) at place i.                   |
| const Col\_amp& at(const int i) | Returns the Col\_amp (i.e. basis vector) at place i.                   |
| const                      |                                                                         |
| void append(Col\_amp Ca)   | Appends a Col\_amp to the basis stored in cb.                          |
| std::string basis\_file\_name() const | Returns a standard filename, used for writing out the basis to a file. |
| void clear()               | Erase the basis, stored in cb.                                         |
Table 11  continued

| Function                                      | Description                                                                 |
|-----------------------------------------------|-----------------------------------------------------------------------------|
| Poly_matr color_gamma(int p1, int p2)         | Function for calculating the color structure part of the soft anomalous dimension matrix. First calculates the effect of gluon exchange on a basis vector and then decomposes the result into the basis. For this to be possible, the basis must clearly contain all resulting basis vectors, meaning for example that it can not be used for Tree_level_gluon_basis. The function is only available for the Trace_basis and the Orthogonal_basis classes. |
| virtual Poly_vec decompose(                   | Each type of color basis has to implement a function for decomposing an amplitude in the color basis. |
| const Col_amp & Ca)                           |                                                                             |
| bool empty() const                            | Is the col_basis empty?                                                     |
| Col_amp exchange_gluon( uint vec, int p1, int p2) | Function for finding the resulting Col_amp after exchanging a gluon between parton p1 and parton p2 in the basis vector vec. |
| bool is_Orthogonal_basis() const              | Is it an Orthogonal_basis?                                                  |
| bool is_Trace_basis()                         | Is it a Trace_basis?                                                       |
| bool is_Tree_level_gluon_basis()              | Is it a Tree_level_gluon_basis?                                             |
| void leading_scalar_product_matrix()          | Finds the leading Nc scalar product matrices, leading_Pspm and leading_dspm. If the polynomial scalar product matrix, Pspm has been calculated, Pspm is used, otherwise Pspm is first calculated and the leading Nc limit is then taken of Pspm. |
| int n_gluon_check() const                     | Returns the number of gluons in the Col_basis after checking that each Col_str in each Col_amp has the same number of gluons. |
| int n_quark_check() const                     | Returns the number of quarks in the Col_basis after checking that each Col_str in each Col_amp has the same number of quarks. |
| virtual void read_in_Col_basis()              | Function for reading in the basis from the default filename (see basis_file_name). |
**virtual void read_in_Col_bas is(**

*std::string filename*)

Function for reading in the basis from the file *filename*. The basis should be in human readable format, of form:

0 \([\{1,3,4,2\}]\)

1 \([\{1,4,3,2\}]\)

2 \([\{1,2\}(3,4)]\)

i.e. first the basis vector number 0,1,2..., etc, then the `Col_amp` corresponding to the basis vector in question.

The `Col_amps` may consist of several `Col_strs`, for example

0 \([\{(1,2,3,4)\}]+[\{(1,4,3,2)\}]\)

1 \([\{(1,2,4,3)\}]+[\{(1,3,4,2)\}]\)

2 \([\{(1,3,4,2)\}]+[\{(1,2,4,3)\}]\)

and each `Col_str` may also contain a `Polynomial`. (The `Polynomial` should multiply the whole `col_str`, rather than a `quark_line`, i.e., the `Polynomial` should stand outside the []-brackets.)

**void read_in_d_spm()**

Reads in a numerical matrix from a file with default filename (see *spm_file_name*) and saves it as a double matrix, *dmatr*, in the member variable `d_spm`. The file should be of the format

\[
\begin{array}{c}
\{d_{11},...,d_{1n}\} \\
... \\
\{d_{n1},...,d_{nn}\}
\end{array}
\]

and may contain comment lines starting with `#` at the top.

**void read_in_d_spm(std::string filename)**

Reads in a numerical matrix from a file *filename* and saves it as a double matrix, *dmatr*, in the member variable `d_spm`. For format, see `read_in_d_spm()`. 

**void read_in_leading_d_spm(std::string filename)**

Reads in a numerical matrix from the file *filename* and saves it as a double matrix, *dmatr*, in the member variable `leading_d_spm`. The format should be as for `read_in_d_spm()`, with 0 for non-diagonal elements.

**void read_in_leading_d_spm()**

Reads in a numerical matrix from a file with default filename (see *spm_file_name*) and saves it as a double matrix, *dmatr*, in the member variable `leading_d_spm`. The format should be as for `read_in_d_spmat()`, with 0 for non-diagonal elements.
### Table 11 continued

| Function | Description |
|----------|-------------|
| `void read_in_leading_P_spm( )` | Reads in a Polynomial matrix from default filename (see `spm_file_name`) and saves it as a `Poly_matr` in the member variable `leading_P_spm`. The format should be as for `void read_in_P_spm()`, with 0 for non-diagonal elements. |
| `void read_in_leading_P_spm(std::string filename)` | Reads in a Polynomial matrix from a file with default filename (see `spm_file_name`) and saves it as a `Poly_matr` in the member variable `leading_P_spm`. The format should be as for `void read_in_P_spm()`, with 0 for non-diagonal elements. |
| `void read_in_P_spm()` | Reads in a Polynomial matrix from a file with default filename (see `spm_file_name`) and saves it as a `Poly_matr` in the member variable `P_spm`. The file should be in the format 

```
{{Poly11,...,Poly1n},
 ..., 
{Polyn1,...,Polynm}},
```

and may contain comment lines starting with `#` at the top. |
| `void read_in_P_spm(std::string filename)` | Reads in a Polynomial matrix from the file `filename` and saves it as a `Poly_matr` in the member variable `P_spm`. The format should be as for `void read_in_P_spm()` |
| `void rename_indices(Col_str & Cs1, Col_str & Cs2) const` | A function to rename the indices in two `Col_strs`, such that in the first they are called 1,2,3..., and in the second the relative order is kept. |
| `virtual Polynomial scalar_product(const Col_amp & Ca1, const Col_amp & Ca2)` | Function for calculating scalar products algebraically using the basis and the scalar product matrix (`Poly_matr P_spm`) in the basis. (Does add implicit conjugated part for `Tree_level_gluon_basis`, as these terms are contained in the matrix of scalar products.) |
| `void scalar_product_matrix()` | Function for calculating the scalar products matrix. This function loops over all basis vectors and stores the value of the scalar product between basis vector i and basis vector j in the i,j -entry in `P_spm` and `d_spm`. The calculation is done using memoization. The symmetry of the scalar product matrix is not used for the calculation, instead it is checked that the resulting matrix is indeed symmetric. |
Table 11  continued

| Function                                      | Description                                                                                                                                 |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| void scalar_product_matrix_num()             | As scalar_product_matrix, but does the calculation numerically. It hence only calculates $d_{\text{spm}}$.                                      |
| virtual void scalar_product_matrix_no_mem()  | As scalar_product_matrix, but without memoization.                                                                                           |
| virtual cnum scalar_product_num(             | Function for calculating scalar products numerically, knowing the basis and the scalar product matrix in numerical form. (Does add implicit conjugated part for $\text{Tree\_level\_gluon\_basis}$, as these terms are contained in the matrix of scalar products.) |
| const Col_amp & Ca1,                          |                                                                                                                                             |
| const Col_amp & Ca2)                           |                                                                                                                                               |
| virtual cnum scalar_product_num(             | Calculates the scalar product between numerical (complex) amplitudes $v_1$, $v_2$ using the numerical scalar product matrix, $d_{\text{spm}}$. The vectors thus needs to be expressed in the basis contained in $\text{cb}$. (Does add implicit conjugated part for $\text{Tree\_level\_gluon\_basis}$, as these terms are contained in the matrix of scalar products.) |
| const cvec & v1, const cvec & v2)             |                                                                                                                                               |
| void scalar_product_matrix_num_no_mem()      | As scalar_product_matrix_num, but without memoization.                                                                                       |
| cnum scalar_product_num_diagonal(             | Calculates the scalar product between numerical (complex) amplitudes $v_1, v_2$ using the numerical scalar product matrix, $d_{\text{spm}}$. Assumes that there are only diagonal contributions. This is useful for calculations in leading $N_c$ limit. (Does add implicit conjugated part for $\text{Tree\_level\_gluon\_basis}$, as these terms are contained in the matrix of scalar products.) |
| const cvec & v1, const cvec & v2)             |                                                                                                                                               |
| void simplify()                               | Simplifies all the basis vectors by using Col_amp.simplify() on the individual Col_amps in the basis.                                           |
| const uint size() const                       | Returns the number of basis vectors.                                                                                                          |
| std::string spm_file_name(                    | Returns a standard filename, used for writing out scalar product matrices. If leading is true, ”l” is appended to the filename. If ”poly” is true ”P.” is added to the filename, and if it is false ”d.”, as in double, is added to the filename. |
| const bool leading, const bool poly) const    |                                                                                                                                               |
| virtual void write_out_col_basis() const     | Function for writing out the basis to a file with default name (see basis_file_name).                                                       |
| virtual void write_out_col_basis(std::string | Function for writing out the basis to a file with name filename.                                                                            |
| filename) const                               |                                                                                                                                               |
Table 11  continued

| Function                                      | Description                                                                 |
|-----------------------------------------------|-----------------------------------------------------------------------------|
| `void write_out_d_spm(std::string filename) const` | Writes out the d_spm to file filename.                                      |
| `void write_out_d_spm() const`                | Writes out d_spm to the standard filename, see spm_file_name.               |
| `void write_out_leading_d_spm(std::string filename) const` | Writes out leading_d_spm to the file filename.                             |
| `void write_out_leading_d_spm() const`        | Writes out leading_d_spm to the standard filename, see spm_file_name.       |
| `void write_out_leading_P_spm(std::string filename) const` | Writes out leading_P_spm to the file filename.                             |
| `void write_out_leading_P_spm() const`        | Writes out leading_P_spm to the standard filename, see spm_file_name.       |
| `void write_out_P_spm(std::string filename) const` | Writes out P_spm to the file filename.                                      |
| `void write_out_P_spm() const`                | Writes out P_spm to the standard filename, see spm_file_name.               |
Table 12  Data members and public functions of the class Trace_type_basis. As Trace_type_basis inherits from Col_basis all the public members of Col_basis are also available, see Table 11

| Data member (protected) | Content |
|------------------------|---------|
| int max ql             | The maximal number of quark-lines allowed in the basis. This is used for constructing bases that only are valid up to a certain order in QCD, such that unused information need not be carried around. |

| Member functions | Effect |
|------------------|--------|
| Trace_type_basis():Col_basis() | Default constructor, uses the Col_basis constructor and sets the variable max ql to 0. |
| Poly_vec decompose(const Col_amp & Ca) | A function for decomposing the color amplitude ca in the basis, returning the result as a Polynomial. |
| std::pair<int, int> new_vector_numbers(const Col_str & Cs, int emitter) | Function for finding the new vector numbers in the new basis (this basis) after radiating a new gluon from the parton emitter in the old color structure is Cs. After emission a linear combination of new basis vectors is obtained. For emission from a quark or an antiquark there is only one resulting color structure, and -1 is returned in the place of the absent color structure. The second vector, where the new gluon is inserted before the emitter comes with a minus sign in the new total amplitude. |
| std::pair<int, int> new_vector_numbers(int old num, int emitter, int n_loop) const | This function is intended for tree-level processes with at most 2 qq-pairs. It finds the new vector numbers in the basis for Nq+Ng+1 partons after radiating a new gluon from the parton emitter. This function does not actually use the cb, but only calculates the basis vector number, which makes it much quicker than the general version. The old vector has number old num, and there were, before emission, Nq quarks (+ Nq antiquarks) and Ng − 1 gluons. For emission from a quark or an antiquark there is only one resulting color structure, and -1 is returned in the place of the absent color structure. The second vector, where the new gluon is inserted before the emitter comes with a minus sign in the new total amplitude. |
Table 13 Public member functions implemented in the class `Trace_basis`. As `Trace_basis` inherits from `Trace_type_basis` all the public members of `Trace_type_basis` and `Col_basis` are also available, see Tables 11 and 12.

| Constructors                  | Effect                                                                 |
|-------------------------------|------------------------------------------------------------------------|
| `Trace_basis()`               | Default constructor, calls the `Trace_type_basis` constructor and sets \( n_q = n_g = 0 \). |
| `Trace_basis(int n_quark, int n_gluon)` | Constructor for creating a trace basis for \( n_{quark} \) \( q\bar{q} \)-pairs and \( n_{gluon} \) gluons. |
| `Trace_basis(int n_quark, int n_gluon, int n_loop)` | Constructor for creating a trace basis for \( n_{quark} \) \( q\bar{q} \)-pairs and \( n_{gluon} \) gluons, keeping only those color structures that can appear to order \( n_{loop} \) in pure QCD. (Note: For electroweak interactions more color structures may be needed.) |

| Member functions              | Effect                                                                 |
|-------------------------------|------------------------------------------------------------------------|
| `void create_basis(int n_q, int n_g)` | Creates a trace basis with basis vectors saved in the `cb` member. Keeps all possible basis vectors, i.e., the basis is valid to any order in perturbation theory. |
| `void create_basis(int n_q, int n_g, int n_loop)` | Creates a trace basis with basis vectors saved in the `cb` member. Keeps only basis vectors consisting of at most \( n_q \) plus \( n_{loop} \) quark lines. |
Public member functions of the class `Tree_level_gluon_basis`.

| Constructor | Effect |
|-------------|--------|
| `Tree_level_gluon_basis()` | Default constructor. |
| `Tree_level_gluon_basis(int n_g)` | Constructor for creating a tree-level gluon basis with \(n_g\) gluons. |

| Member functions | Effect |
|------------------|--------|
| `void create_basis(int n_g)` | Creates a basis with basis vectors saved in the `cb` member. Each basis vector is a sum of two traces, of form \(\text{tr}[t^a t^b \ldots t^z] + (-1)^{N_g}\text{tr}[t^z \ldots t^b t^a]\). The charge conjugated trace is implicit, and only one trace is actually carried around which speeds up calculations. |
| `void read_in_Col_basis()` | Function reading in the basis from default name (see `basis_file_name` in the `Col_basis` class). The full basis, including the charge conjugated part should be contained in the file. (This is to simplify comparison with other programs, such as ColorMath.) |
| `void read_in_Col_basis(std::string filename)` | Function for reading in the basis from a file. The file should contain the whole basis, including the charge conjugated part. |
| `void write_out_Col_basis()` const | Function for writing out the basis to default name, (see `basis_file_name` in the `Col_basis` class). The full basis, including the charge conjugated part is written out. (This is to simplify comparison with other programs, such as ColorMath.) |
| `void write_out_Col_basis(std::string filename)` const | Function for writing out the basis to `filename`, (see `basis_file_name` in the `Col_basis` class). The full basis, including the charge conjugated part is written out. (This is to simplify comparison with other programs, such as ColorMath.) |
Table 15 Public member functions of the class Orthogonal_basis. Orthogonal_basis inherits directly from Col_basis, meaning that public members of Col_basis are also available, see Table 11

| Data members       | Content                                                                 |
|--------------------|-------------------------------------------------------------------------|
| dvec diagonal_dspm | To contain information about scalar products as a dvec, i.e., entry i is the square of vector i. |
| Poly_vec diagonal_Pspm | To contain information about scalar products as a Poly_vec, i.e., entry i is the square of vector i. |

| Constructor                               | Effect                                                                 |
|-------------------------------------------|-------------------------------------------------------------------------|
| Orthogonal_basis():Col_basis()            | Default constructor, puts private variable orthogonal_basis=true and calls the constructor of Col_basis. |

| Member functions                          | Effect                                                                 |
|-------------------------------------------|-------------------------------------------------------------------------|
| Poly_vec decompose(const Col_amp & Ca)    | The decomposition of a Col_amp in an orthogonal basis is done by calculating scalar products and dividing out the norm. The norm is evaluated numerically. |
| void diagonal_scalar_product_matrix(     | Calculates the diagonal entries in the scalar product matrix, and (depending on arguments), saves them to the member variables diagonal_Pspm and diagonal_dspm. This function is used by the Orthogonal_basis version of diagonal_scalar_product_matrix. |
|   bool save_P_diagonalspm, bool           |                                                                         |
|   save_d_diagonalspm, bool use_mem)        |                                                                         |
| std::string diagonalspm_file_name(       | Creates a default filename for writing out diagonal scalar products. The boolean variable leading should be true if the name is for a leading Nc variable. The filename is then modified accordingly. |
|   const bool leading, const bool poly)    |                                                                         |
| Polynomial scalar_product(const Col_amp & | Function for calculating scalar products given the information about the basis and the scalar product matrix in the basis. The Col_amps are first decomposed using decompose, and then squared using the scalar product matrix Pspm. An orthogonal scalar product matrix is assumed. |
|   Ca1, const Col_amp & Ca2)               |                                                                         |
| void scalar_product_matrix()              | Calculates the scalar product matrix assuming the basis to be orthogonal. Calculates both the double (dspm) and the Polynomial (Pspm) matrices and saves to default filenames. |
| cnum scalar_product_num(const Col_amp &   | Function for calculating scalar products given the information about the basis and the scalar product matrix in numerical form. The Col_amps are first decomposed using decompose, and then squared using diagonal_dspm. For Orthogonal_basis an orthogonal scalar product matrix is assumed. |
|   Ca1, const Col_amp & Ca2)               |                                                                         |
Table 15 continued

| Function | Description |
|----------|-------------|
| `cnum scalar_product_num(const cvec & v1, const cvec & v2)` | Calculates the scalar product between decomposed amplitudes v1, v2 using the `diagonal_d_spm` diagonal numerical scalar product matrix. The vectors need to be expressed in the basis contained in `cb`, i.e., the decomposition has to be known. |
| `void write_out_diagonal_d_spm(std::string filename) const` | Writes out `diagonal_d_spm` to the file `filename`. |
| `void write_out_diagonal_d_spm()` const | Writes out `diagonal_d_spm` to the standard filename, see `diagonalspm_file_name`. |
| `void write_out_diagonal_P_spm(std::string filename) const` | Writes out `diagonal_P_spm` to the file `filename`. |
| `void write_out_diagonal_P_spm()` const | Writes out `diagonal_P_spm` to the standard filename, see `diagonalspm_file_name`. |
| `void write_out_diagonal_spm(const dvec & dv, const bool leading) const` | Function for writing out a `dvec` (the diagonal scalar products, `diagonal_d_spm`) to a file with standard filename given by `diagonalspm_file_name`. The boolean variable `leading` should be true if `dv` only has leading `Nc` contributions. The filename is then modified accordingly. |
| `void write_out_diagonal_spm(const Poly_vec & pv, const bool leading) const` | Function for writing out a `Poly_vec` (the diagonal scalar products, `diagonal_d_spm`) to a file with standard filename given by `diagonalspm_file_name`. The boolean variable `leading` should be true if `Poly_vec` only has leading `Nc` contributions. The filename is then modified accordingly. |
References

1. R. Mertig, M. Bohm, A. Denner, FEYN CALC: computer algebraic calculation of Feynman amplitudes. Comput. Phys. Commun. 64, 345–359 (1991)

2. J. Hakkinen, H. Kharraziha, COLOR: a computer program for QCD color factor calculations. Comput. Phys. Commun. 100, 311–321 (1997). arXiv:hep-ph/9603229

3. T. van Ritbergen, A. Schellekens, J. Vermaseren, Group theory factors for Feynman diagrams. Int. J. Mod. Phys. A 14, 41–96 (1999). arXiv:hep-ph/9802376

4. M. Sjodahl, ColorMath—a package for color summed calculations in SU(Nc). Eur. Phys. J. C 73, 2310 (2013). arXiv:1211.2099

5. J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, T. Stelzer, MadGraph 5: going beyond. JHEP 1106, 128 (2011). arXiv:1106.0522

6. M. Bahr et al., Herwig++ physics and manual. Eur. Phys. J. C 58, 639–707 (2008), arXiv:0803.0883

7. J. Bellm, S. Gieseke, D. Grellscheid, A. Papaefstathiou, S. Platzer, et al., Herwig++ 2.7 Release Note. arXiv:1310.6877

8. I.E. Paton, H.-M. Chan, Generalized Veneziano model with isospin. Nucl. Phys. B 10, 516–520 (1969)

9. P. Dittmer, Invariant tensors in SU(3). II. Commun. Math. Phys. 27, 44–52 (1972)

10. P. Cvitanović, Group theory for Feynman diagrams in non-Abelian gauge theories. Phys. Rev. D 14, 1536–1553 (1976)

11. P. Cvitanović, P. Lauwers, P. Scharbach, Gauge invariance structure of quantum chromodynamics. Nucl. Phys. B 186, 165–186 (1981)

12. F.A. Berends, W. Giele, The six gluon process as an example of Weyl–Van Der Waerden spinor calculus. Nucl. Phys. B 294, 700 (1987)

13. M.L. Mangano, S.J. Parke, Z. Xu, Duality and multi-gluon scattering. Nucl. Phys. B 298, 653 (1988)

14. M.L. Mangano, The color structure of gluon emission. Nucl. Phys. B 309, 461 (1988)

15. Z. Nagy, D.E. Soper, Parton showers with quantum interference. JHEP 09, 114 (2007). arXiv:0706.0017

16. M. Sjodahl, Color structure for soft gluon resummation—a general recipe. JHEP 0909, 087 (2009). arXiv:0906.1121

17. S. Keppeler, M. Sjodahl, Orthogonal multiplet bases in SU(Nc) color space. JHEP 1209, 124 (2012). arXiv:1207.0609

18. M. Sjodahl, J. Thoren, Decomposing color structure into multiplet bases, work in preparation

19. F. Maltoni, K. Paul, T. Stelzer, S. Willenbrock, Color flow decomposition of QCD amplitudes. Phys. Rev. D 67, 014026 (2003). arXiv:hep-ph/0209271

20. S. Platzer, S. Gieseke, Dipole showers and automated NLO matching in Herwig++. Eur. Phys. J. C 72, 2187 (2012). arXiv:1109.6256

21. Y.-J. Du, M. Sjödahl, J. Thorén, Recursion in multiplet bases for tree-level MHV gluon amplitudes. arXiv:1503.0053

22. E. Gerwick, S. Hoeche, S. Marzani, S. Schumann, Soft evolution of multi-jet final states. arXiv:1411.7325

23. S. Platzer, M. Sjodahl, Subleading Nc improved parton showers. JHEP 1207, 042 (2012). arXiv:1201.0260

24. S. Platzer, Summing large-Nc towers in colour flow evolution. Eur. Phys. J. C 74(6), 2907 (2014). arXiv:1312.2448

25. F. Campanario, T.M. Figy, S. Platzer, M. Sjodahl, Electroweak Higgs boson plus three jet production at next-to-leading-order QCD. Phys. Rev. Lett. 111(21), 211802 (2013). arXiv:1308.2932

26. P. Cvitanović, Group Theory: Birdtracks, Lie’s, and Exceptional Groups. Princeton University Press, Princeton (2008). http://www.birdtracks.eu