An introduction to Qgraf 4.0

P. Nogueira
CeFEMA, Instituto Superior Técnico
Universidade de Lisboa (ULisboa)
Lisbon, Portugal

Abstract

This document[1] is a (nearly) unified guide for `qgraf-4.0` that includes revised and updated versions of the previously available documentation. For instance, an extended discussion on the *generic* sign convention implemented in the program (which does not rely on graphical rules) is included. On the other hand, some other details (references, etc) will be added later. As usual, a summary of the new features is included in the Changelog (the last section).

The latest pack includes some (compilable) examples that illustrate how the new API (ie application programming interface) works. The latest version might still not be the definitive `qgraf-4.0` — rather just an advanced development version that still has to go through further testing, at least — but it is getting close now.

[1] This version is part of the `qgraf-4.0.5` pack (September 2024), and was typeset in TEX aided by the `Eplain`, `epsf`, `colordvi` and `manmac` packages.

Contents

0. Preliminary remarks, current plans

This guide includes up-to-date examples of a control-file (Section 22) and of a model-file (Section 27). Moreover, Section 2.1 shows how to create an executable file, or binary, for use in auto-mode (the usual mode, in which the program runs 'autonomously', supported by the operating system and some installed libraries); note that an additional compiler option has just become necessary. The recently introduced API-mode, where some other program calls QGRAF by means of an interface, is discussed in Section 37, and the corresponding compilation procedure in Section 2.4.

In recent years, most of the development has been increasingly focused on usability rather than on diagram related features. The corresponding releases have been closer to development versions than in the past, for that allows adding new features more frequently. Nevertheless, that development phase seems to be nearing its end, and the above mentioned trend will end with it — new releases are likely to become either less frequent or include just the odd new feature. Obviously, there are still various non-trivial features, some of which suggestions, waiting for their turn.

Actually, `qgraf-4` will rely on revised, slightly different *input specifications*, and its use in automatic set-ups is not recommended for the time being; some ancillary software will be available, to help with the corresponding transition (Section 41). The list of changes follows, in the form of 'rules' (rules 4–7 are new but rule 10, which has also been added to the list, is not); a few of these changes have been hinted at for some time. Whenever possible, any such change should be accompanied by a corresponding feature of the file conversion tool, so as to render the number of changes much less relevant.

(*1*) the names of files and directories will be constrained — as described in Section 4.2 and in Section 7.3, only generally;

(*2*) only the characters '`%`' and '`#`' will be allowed as annotation-marks (the asterisk will be excluded at last); only one type of annotation-mark will be allowed in any given file;

(*3*) some 'void' control-file statements will no longer be accepted (Section 5.2);

(*4*) the default external momenta and the default prefix of the integration momenta will be dropped (the respective identifiers will have to be declared, should they be needed);

(*5*) the default filename for the control-file, ie `qgraf.dat`, will be deleted (a filename will have to be specified, always);

(*6*) the style-keyword `<diagram_index>` will be re-defined in the epilogue section, where it should be replaced by `<diagram_counter>` (Section 32.6);

(*7*) the style-keywords `<command_loop>` and `<command_line_loop>` will be renamed (respectively) `<statement_loop>` and `<statement_sub_loop>`; the style-keyword `<command_data>` will be replaced by `<sub_statement>`, which is not fully equivalent;

(*8*) in model-files, both constants and functions must be declared; any function involved in a numerical constraint (eg a `vsum` statement) should be declared `integer` as well;

(*9*) the implicit statement continuation will be dropped, and a statement continuation mark will be required in model-files and control-files, in every appropriate instance;

(*10*) for any input file, the maximum allowed length of any line will be increased to 120.

The set of backward-incompatibilities for `qgraf-4` should now be considered closed. It ought to be clear that the option to do a 'rather good sweeping' prevailed, which included the deletion of some unnecessary defaults. Actually, it is likely that a good number of the above rules are broken only rarely in current practice, and that little (if anything) has to be done about them (ie these). Rules 1–5 *have* been implemented already, and the remaining rules should become mandatory with the next stable version (which will probably be `qgraf-4.0`, sometime in mid or late 2025). Rule 10 would not be a (new) restriction by itself had it not been tied to rule 9; neither one has been fully enforced thus far so as to keep current input files (specially model-files) compatible for a little while longer.

A second look at the above list ought to suggest that its impact should be much more limited than the one that might be suggested by the number of items alone. Rules 1–5 consist in fairly simple restrictions that could be implemented (eg) at once in related third-party software since compatibility with currently available versions of the program would not be affected. Rule 8 could also be implemented in software relying on `qgraf-3.6`.

At present, the file conversion tool can deal with rules 2 and 9 (and, trivially, with 'non-rule' 10). What will not be addressed, for obvious reasons, includes the modification of third-party software dealing with the construction and naming of input files (rules 1 and 5 are the ones falling into this category). As already mentioned, however, these two rules could be implemented now, with no 'side-effects' (regarding QGRAF), even in software relying on `qgraf-3.4` — so that transitioning to `qgraf-4` later on would be simpler.

The ability to deal with the remaining rules (3–4 and 6–9) might be implemented in that tool also (though later, possibly with the release of the patch version that precedes the next attribution of *stable* status), depending on the time available. Regarding the changes already implemented (rules 1–5), the automatic conversion of model-files and style-files is sufficient; the conversion of the control-file is not. Rules 6–8 should be addressed, so that in the worst case it would all resume to creating the control-file and dealing with filenames.

The reasons for those changes include the following: fixing or improving poor design choices (rule 9, specially), improving compatibility with different operating systems, trying to ensure that input files can be easily interpretable, and eliminating potential pitfalls. There is also a reason for introducing those changes at this time. That reason is the relative stability of both that interface and of QGRAF-R. For example, QGRAF and QGRAF-R should in some 'near-future' be able to read exactly the same model-files (including those that use the language extensions introduced with `qgraf-3.6`), as well as control-files that satisfy similar specifications. In principle, the induced disruption should be minimized by introducing all those changes simultaneously while providing some software to help with the file conversion (as opposed to more than one set of changes, introduced at different times). Nevertheless, since 'simultaneity' is not really practical at this point (it would require suspending new releases for quite a while), the next best thing seems to be providing some ancillary software, which should evolve during the transition period (Section 41).

As with any feature introduction, users that rely (or plan to rely) on packages that depend on this program may have to postpone the effective use of (some of) the new features. As a possibility/suggestion, there might be two distinct directories for model-files — say, `models3/` and `models4/` — with `models3/` storing the files in the format accepted by `qgraf-3.6`, and `models4/` the newer version of those files, to be accepted by `qgraf-4`. In fact, until things stabilize, a model-file in `models4/` could be created on-the-spot from a file in `models3/`, whenever necessary, using QGRAF itself (Section 41). That possibility applies also to style-files. Thus, apart from an automatic translation involving a few very brief runs, current input files could continue to be used until the time for the full transition arrives.

Regarding the previously announced interface (API), there are some other changes and additions; the 'final' specification is presented in Section 37; Although further testing is necessary, it should be possible to use the latest version for related software development.

Some of the past developments are due to *'suggestion-requests'* from various users. Concerning the more recent features (ie introduced with `qgraf-3.1.5` and later versions), Lance Dixon suggested something like the diagram option `onshellx`, and Vladyslav Shtabovenko suggested having more than one kind of output-file in the same run, allowing the name of the control-file to be definable, and a file overwriting feature (and, indirectly, allowing the use of some additional characters in input files, in annotations). Regarding run-time optimizations, John Gracey and V. Shtabovenko suggested improving the efficiency of (respectively) the graph generation for 'large' orders of perturbation theory, and of 'additive-filters' such as `vsum`. The implementation of a formal way to run the program as a sub-program was inspired in a not so recent request of Mikhail Tentyukov. The suggestions to have no display-output (or a very limited one) and to extend the `loops` statement are likely due to Jos Vermaseren, some time in the nineties!

0.1 Brief notes on the terminology

In what follows, the *display* means the standard output (redefined or not), and *to display 'something'* means to send a graphical representation of 'something' (using ASCII characters, that is) to the standard output — computer monitor or otherwise. The information sent to the standard output may also be dubbed the *display-output* (as opposed to the output that goes into some file).

Moreover, we shall say that QGRAF is in *auto-mode* when running autonomously (ie supported by the operating system and the standard libraries only, and usually launched from the command-line), and that it is in API-*mode* when being called from another program using the API (application programming interface) described in Section 37.

Part I — Introduction

1. About the program

QGRAF is a Feynman diagram generator — more precisely, a (Fortran) computer program designed with the aim of generating *symbolic* descriptions of Feynman diagrams in various types of QFT models — with a configurable (ie programmable) output. Its main algorithm, of orderly type, is based on the method described in [1].

The program[2] was created to automate the often tedious (and error prone) task of writing down symbolic amplitudes for scattering processes, specially when the number of diagrams involved is large. It is aimed mainly at writing the amplitudes in terms of products of the matrix elements that follow directly from the Feynman rules. By presenting the amplitudes in that kind of *'raw'* form, and also by allowing several types of constraints to be imposed on the generated diagrams, it should hopefully provide a common starting point for various types of calculations.

The symmetry factor, a sign[3] and a set of momenta should be provided for each diagram; no other field theoretic computation is performed, however. It should be noted that the *generic* sign convention implemented in the program does not rely on graphical rules.

Neither vacuum diagrams (ie diagrams with no external fields) nor non-connected diagrams are generated; in addition, diagrams must have at least one vertex, which means that tree-level propagators are not 'generated' either.

The program was first used in a (then cutting-edge) 3-loop massless QCD calculation, circa 1995. Since then, owing not only to impressive developments in multi-loop computational methods but also to sustained advances in computer technology, quite a few calculations involving many thousands of diagrams have been reported in the physics literature.

The evolution of QGRAF, measured in terms of the number of lines of code, is depicted in Fig. 1. While most of the source code consists in Fortran code, some preprocessor code is now intermingled, and there is even some C code (for the header file). Obviously, these numbers do not tell the whole story — not every line of code survives from one version to the next.

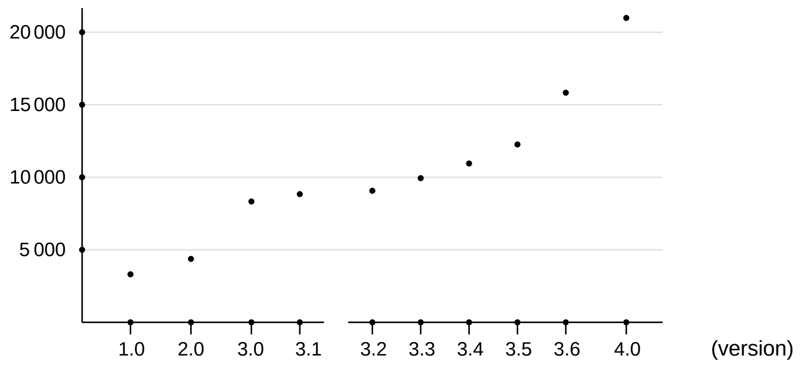

Fig. 1. The approximate number of lines of code.

[2] In this document, *'the program'* shall mean QGRAF.

[3] One that derives from the anti-commutation relations, naturally.

2. Downloading, compiling and installing

The current URL for the program's website is

```
http://cefema-gt.tecnico.ulisboa.pt/~paulo/qgraf.html
```

and the one for automatic downloads (Section 40) is

```
http://qgraf.tecnico.ulisboa.pt/
```

The (old) domain `ist.ult.pt` is likely to become obsolete at some point.

In that website may also be found the disclaimer and the general license of the program. Additional terms might be found in the header of the respective Fortran file(s).

The source files of recent versions (starting with `qgraf-3.5`) are expected to conform to the Fortran 2008 standard[4] (and seemingly the Fortran 2018 standard as well) although testing has been restricted to the GFortran compiler. Note that there are executable/binary versions of GFortran for several operating systems, as described in

```
https://gcc.gnu.org/wiki/GFortranBinaries
```

but most Linux distros have ready-to-install packages, which might be installed in your computer already (more than one package may be needed).

2.1 Compiling and linking (auto-mode)

Starting with `qgraf-4.0.5` the source code will include preprocessor directives, which have to be addressed during the compilation. Regarding `qgraf-4`, in a Linux/GNU system one may (eg) copy the source file to an empty directory and execute the commands

```
\mkdir fmodules
\gfortran -cpp -o qgraf -Os -J fmodules qgraf-4.0.5.f08
```

to obtain an executable named `qgraf` (the correct version numbers should be used, of course); the option `-cpp` will be required from now on, to enable preprocessing. Seemingly, that option can be used with `qgraf-3` without producing any effect since in this case the source code does not include any preprocessing directive. There is a minor inconvenience in that a number of Fortran module files are created (one file for each module defined in the source code); nevertheless, GFortran can place those files in a user-definable directory (option `-J`), and they can be deleted once an executable file has been created.

The option `-Os` tells the compiler to try to minimize the size of the executable while still enabling some optimizations that tend to increase the performance of the generated code; option `-O2` optimizes a bit more (for performance) than `-Os`, and `-O3` is also a possibility (option `-Ofast` should *not* be used).

For the latest package, executing the command

```
\make qgraf
```

in the directory that contains the `Makefile` also produces an executable file.

If the compilation produces an error message like this one

```
Fatal Error: Cannot read module file 'qaski.mod' opened at (1),
  because it was created by a different version of GNU Fortran
```

[4] Not only that, a couple of features of Fortran 2008 are actually used — hence those recent versions do not conform to any preceding standard, eg to Fortran 2003.

that likely means that the compiler found some module files (which it might have ignored, or even deleted) in a format it cannot understand, and then it got a bit confused. In that case, find and delete all the problematic module files, *both* in the working directory and in the directory specified with the option `-J` (their filenames should start with the character '`q`' and, as reported by the compiler, end with '`.mod`'). Better yet, remove the previous Fortran module files, if any (in either directory), before re-compiling.

In fact, the existence of earlier module files (in the same format, but for a different version of the program) can lead to another type of problem, namely compilation failure due to apparent programming errors! So... any pre-existing Fortran module files should always be deleted before compiling the source code, or else a new directory (and a subdirectory) should be created for that purpose, as already suggested.

As there are no modules in Fortran 77, the next command can be used as a template for compiling older versions (that is, `qgraf-3.4` and earlier versions).

```
gfortran -o qgraf -Os qgraf-3.4.2.f
```

Compiling with options that rely on somewhat sophisticated code transformations to produce more efficient executable files (this is often called *aggressive optimization*), such as `-O3` in GFortran, should probably be avoided unless the additional speed is really needed — compilers, like other computer programs, are not immune to errors. For relevant computations (ie other than testing), such options should be used only after some preliminary successful testing (eg with similar but computationally less-demanding cases), where the outputs of two or more executables compiled with different optimization levels are compared — say, an executable with the desired 'aggressive' level, and one or more executables with 'non-aggressive' levels. Furthermore, it should be kept in mind that changing the program's environment (either hardware or software) can invalidate a previous test — for example updating the compiler, which one's computer might do automatically, and then recompiling.

Concerning INTEGER type variables (in Fortran), QGRAF requires 32-bit integers in auto-mode and 64-bit in API-mode (which should be supported by any current compiler).

It is perhaps worth mentioning that the program does not make (explicit) operating system calls to run shell commands; in fact, it is not even aware of which operating system is being run. Thus, neither of the following types of statements is used.

```
call system(...)
call execute_command_line(...)
```

The first is a non-standard system call available with GFortran (eg for Fortran 77), and the other is the standard Fortran call, introduced with Fortran 2003. Naturally, (Fortran) output and input statements are used extensively, providing access to the standard output, to the control-file, and to any other file specified in the control-file.

2.2 Internal parameters

It is not advisable to change the default value of most of the internal parameters, as there is a real risk of ending up with a defective program (specially for `qgraf-3.1.5` and more recent versions). The only changes that should be 'safe' are the ones that consist in increasing the values of one or more of the following parameters

```
scbuff     sibuff     swbuff0
```

should the program report that they are too small (these parameters control the size of some parts of the working memory). That should not be a frequent occurrence: there would have to be (eg) an unusually large input file, or a model with a large number of vertices (or at least some vertices of high degree). One may then double the value of each reported parameter, more than once if necessary, until there is no error (alas, upper bounds also exist); values of the form $2^n - 1$ should be preferred.

2.3 Installing (auto-mode)

The installation is also system dependent but does not require one to perform some particularly complex procedure. In a Linux/GNU based system one may copy the executable file to (eg) one of the directories

```
/usr/local/bin/
/opt/bin/
```

(possibly after creating it, should it not exist), and then make sure that that executable has the correct file permissions. For example, if the executable produced by the compilation procedure is in the current directory (and is named `qgraf`), one could do the following (as the `root` user, say, or using the `sudo` command).

```
\cp -i qgraf /usr/local/bin/
\chmod 755 /usr/local/bin/qgraf
```

After that, if necessary, each user would have to create an appropriate alias for `qgraf`, or else add the full name of the installation directory to its own `PATH` (the shell variable). That procedure corresponds to a system-wide installation. Should a single-user installation suffice, then the installation directory could be

```
$HOME/bin/
```

Actually, since it is usually convenient to have a dedicated directory for the program (to store all the necessary files, including subdirectories for model-files and style-files) the executable could be moved to (or possibly left in) that directory, and then run from there as (eg)

```
./qgraf my_control_file
```

2.4 Compiling (API-mode)

Generating an object file suitable for API-mode is also simple if the `Makefile` that is part of the package (can be and) is used, but the exact command depends on which interface subroutine is chosen. The currently available subroutines are the following.

```
qinterf2008f
qinterf2008c
qinterf2018c
```

A Fortran 2008 program should call subroutine `qinterf2008f`. The compilation step might consist in something like this:

```
gfortran -c (...) -Os -J fmodules qgraf-4.0.5.f
gfortran (...) -o x -Os -J fmodules qgraf-4.0.5.o x.f08
```

Here, `x.f08` is a Fortran program that calls the interface, `x` is the final binary or executable, and '(...)' means an appropriate set of command-line options (see the `Makefile`, and in particular how the binary `pf08` is created).

Recall that the conformance of a computer program to some (revised) standard depends usually on using neither deleted features nor non-standard extensions — hence (eg) a Fortran 77 program might be a perfectly valid Fortran 2018 program (but possibly not a very good one). Thus, a Fortran 95 program will likely work fine with QGRAF after being 'polished' into a Fortran 2008 conforming program (if it is presently non-conforming, in which case the compilation procedure described in the `Makefile` should produce error messages). Whether or not the polishing step can be skipped by adjusting the Fortran standard on the final compilation, is something that has not been tested; moreover, that option might be compiler dependent.

The latest package includes also some actual examples of interfacing, whose compilation relies on a special set of preprocessor and compiler options. The included `Makefile` can deal directly with each one of those examples; every example uses the same source file for QGRAF but the preprocessing options differ. Naturally, it is possible to look inside that file and copy (and possibly adapt) any command included in it.

In the previous example, the compilation has been split into two steps to isolate the origin of any problem that might arise. The previous warnings about Fortran module files should be kept in mind. The `Makefile` includes some commands for deleting those files but some have been commented out; the deletion of the respective annotation signs (a leading '`#`' in every case) should be considered once any required adjustments have been made.

When mixed-programming is involved (for example, when QGRAF is called by a C program, which is in fact the only mixed-programming case to be considered) the compilation will very likely require two steps, the first using a Fortran compiler (`gfortran`, say) and the second step a C compiler (eg `gcc`). The examples based on the files `pxq08.c` and `pxq.c` illustrate the two mixed-programming interfaces implemented in `qgraf-4.0`. Note that there is an header file (ie `qgraf.h`) to be imported by any C program using the interface.

In the former example (ie `pxq08.c`), the interoperability between the two languages is based on the standard tool available for Fortran 2008 — ie the `iso_c_binding` module, used on the Fortran side. In this case, the C program should call subroutine `qinterf2008c`.

In the second example (ie `pxq.c`), a more recent standard based on *C-descriptors* is used as well — that is, the header file `ISO_Fortran_binding.h` is used on the C side. In the supplied `Makefile`, the compilation that creates the binary `pxq` uses the Fortran 2018 and C17 standards. Now, the C program should call subroutine `qinterf2018c`.

3. Files, records, and characters

QGRAF deals with several types of files, whose roles are explained below. The name of the control-file has to be specified when launching or initializing the program — in auto-mode, as a command-line argument (Section 35). The other filenames should be declared in the control-file using appropriate statements.

- `control-file`

  The input file that contains the main instructions for the program.

- `message-file`

  The output file where warning and error messages should be saved.

- `model-file`

  The input file that describes the model to be used in the diagram generation.

- `output-file`

  The main type of output file, where the descriptions of the generated diagrams should be saved.

- `style-file`

  An input file that contains the instructions for generating an output-file, and in particular for describing each generated diagram.

In this section we shall be concerned mainly with the input files, whether created by the user or automatically generated. Moreover, the focus will be on their basic properties, such as the character set to be used, the size of the records (ie lines) in those files, annotations and so on.

Since there are three types of input files, one might say that QGRAF 'speaks' three languages — it certainly parses them. Both the control-file and the model-file consist of a sequence of statements (and possibly comments) but the style-file looks a bit different.

In what follows, and depending on the context, the word *filename* may refer to the name of (ie character string defining) a regular file, directory, or symlink.

3.1 The character set

First of all, the character set should be the (7-bit) ASCII (nowadays a very small subset of UTF-8) which includes 95 printable characters and 33 control (non-printable) characters. The printable characters consist of 26 letters (upper and lower case), 10 decimal digits and 33 punctuation symbols. The control characters that remain in use[5] include (eg) the characters that mark the end of the line (or record) or the end of the file, but their exact list depends on the operating system (and possibly even file system). Hereafter, with the exception of *'newline'*, we will usually do our best to pretend they are not there.

The control characters remain typically in the background, so that the user needs not deal with them explicitly (nor even see them). An exception to that rule is discussed in Section 35.3, though.

[5] In fact, many of those 33 control characters are no longer used.

3.2 The space character

Apart from being able to improve readability, the spaces in input files either act as a separator, delimiting character (sub-)strings, or represent real space characters in some ('encoded') strings (Section 4.1). Often, they are not even needed as a separator since other punctuation symbols can also fulfil that role.

3.3 The annotation-mark(s)

In control-files and model-files, any line whose first character is either '%' (percent sign) or '#' (hash mark) is ignored. Thus, that kind of lines (to be referred to as *annotations*) can be used for explanatory notes and other comments. No annotation should 'break' a statement extending over two or more lines. Style-files may also include annotations, but the restrictions are different (Section 29.1).

Only one type of annotation-mark (ie '%' and '#', when employed in the above described role) can be used in any given input file.

3.4 The statement continuation mark, wide(r) files

The input files may contain lines (or records) with up to 120 characters,[6] not counting *newline*. Simultaneously, for model-files and control-files (but not for style-files), a statement continuation mark (the backslash '\') has been introduced. This feature does not apply to annotations — only statements can be continued. Two examples of control-file statements using the new specification follow.

```
output_dir = \
   '/tmp/my_temporary_dir/' ;

partition = 3^2\
               4^1 ;
```

A backslash character should appear at the rightmost position of each appropriate input line and should not break identifiers, keywords, or other special strings. Each statement should occupy either a single line or a number of consecutive lines; in addition, continuation lines should not be 'empty', ie each one has to include some relevant part of the statement, however small (a single punctuation mark, even).

The number of lines occupied by a single statement is also bounded since there is a maximum allowed statement length (not smaller than 500).

When no 'long lines' exist (and until the next stable version is released), statement continuation marks may or may not be used but there must be some consistency — in any such file, they should be used either in every appropriate instance or in none.

NB: Although not particularly difficult, the addition of statement continuation marks to model-files and control-files can be done with QGRAF (Section 41).

[6] But only up to 80 characters with previous versions.

4. Basic types of character strings

This section discusses some of the types of character strings recognized by the program, namely identifiers, integers, rationals and encoded strings. Naturally, several punctuation marks are also recognized, but those will be presented when needed, without further ado (note that some of those marks are not single ASCII characters, while others which are single ASCII characters are not usually considered as punctuation marks). Moreover, the program recognizes (*i*) a special type of strings that appear in the style-file (the 'style-keywords'), to be discussed later, and (*ii*) the signs '`+`' and '`-`' when not part of integers and rationals (in model-files).

- An *identifier* is a string made of letters, digits, and underscores, with the condition that the first character is a letter. Examples: `spin`, `F_0`, `csi___`.
- An *integer* is a sequence of digits, possibly preceded by either a plus or a minus sign. Examples: `-0`, `+17`, `1234`.
- A *rational number* is either an integer or a sequential concatenation of (*i*) an integer, (*ii*) a slash '`/`', and (*iii*) a nonzero unsigned integer. Examples: `1/3`, `-1`, `+0/5`.

The characters mentioned in each of the above definitions are the only ones allowed. In particular, space characters are not allowed in those strings, although they can be used in a different type of string that will be described in the next paragraphs. When dealing with strings that include spaces it is often convenient to use the symbol ␣ — the so called visible space [2]. This symbol allows one to clarify whenever a space is indeed part of a string, or where a record does begin, or end (specially if there are leading and/or trailing spaces).

4.1 Encoded strings

It would be convenient to have strings that are accepted as indivisible, even if they included sub-strings that could be classified as identifiers, integers, punctuation marks, and so on. For example, suppose that the string

```
Quantum␣Electrodynamics␣in␣D=4-epsilon␣dimensions
```

should be accepted as a single object. A solution to that problem consists in encoding the original string into another one that cannot be mistaken for other types of strings (or collection thereof), and then supply the encoded string to QGRAF, which will simply decode it. The ASCII character 39 (which usually represents the apostrophe or the right quote) plays a special role in the encoding algorithm that was implemented. To encode a string is straightforward: in the first place every apostrophe is duplicated (ie replaced by two consecutive ones) and then a further apostrophe is added to each end of the resulting string. The encoded form of the above string is simply

```
'Quantum␣Electrodynamics␣in␣D=4-epsilon␣dimensions'
```

Here are other examples: the empty string is encoded into the string `''` of length 2, and a single apostrophe (when considered as a string by itself) is encoded into the string `''''` of length 4. This type of encoding (to be dubbed *normal-encoding*) is also used in Fortran programs, as known, but the syntax for string concatenation is different (see below).

As stated earlier, there is an explicit upper bound for the record size. This condition places an effective bound on the size of (eg) identifiers, since an identifier cannot be broken into components. Nevertheless, strings presented to the program in encoded form can avoid that bound: before being encoded, any such string can be decomposed into several components, and then the encoded strings corresponding to those components may be written sequentially, either on the same line (separated by at least one space!) or on a number of consecutive lines. For example, the string

```
'f(x)='␣␣'1+sin(x)'
```

is not the normal-encoding of any (single) string, but it will be decoded as

```
f(x)=1+sin(x)
```

since the concatenation is implicit; by contrast, the string

```
'f(x)=''1+sin(x)'
```

will be decoded as

```
f(x)='1+sin(x)
```

because in this case there is a single component, not two.

A given string may have many (non-normal) encodings; we will say that a string t is an encoding of another string s if and only if t will be decoded as s.

4.2 Constraints on filenames and paths

The names of files and directories (and paths) accepted by the program are encoded strings. The following restrictions apply to their non-encoded forms, whether specified in the control-file or in the command-line:

- each (ASCII) character should be either alphanumeric (ie a letter or a digit) or one of three 'special characters' — namely, the *underscore*, the *hyphen-minus*, and the *dot* (or *period*);
- the first and the last characters should be alphanumeric;
- no two special characters should appear consecutively.

Moreover, these constraints apply to each path component (the constraints for the path separator are described in Section 7.3). For example, the following statements (which define the output-directory and an output-file, respectively) are valid.

```
output_dir = '/home/user/qgraf/tmp_dir/' ;
output = 'tmp_file-1.out' ;
```

The latter statement includes a filename with every possible type of special character, as defined previously.

The above mentioned restrictions may be bypassed at the operating system level with the help of symlinks. The `noblanks` config option might be useful for declaring filenames in the control-file.

4.3 A constraint on integer numbers

Arbitrarily long integers are not supported (nor arbitrarily long rationals, therefore). There is then a largest (but not very large, positive) integer M supported, in the sense that any integer z — either read, written, or even evaluated by the program, excluding the diagram index and the index offset — should satisfy the condition

$$|z| \leq M.$$

Presently (for `qgraf-4.0`), $M \sim 2^{31/2} > 10^4$. If the internal parameters are not modified, that value seems to be more than enough for the currently required tasks — eg labelling the various 'elements' of the diagram, generating indices specified in the style-file, and evaluating both symmetry factors and other weights (for the 'filters' `psum` and `vsum`).

It is possible to specify larger 'integers' in the model-file, but only in encoded form. Thus, in practice, those would not be integers — nonetheless they can be reproduced (unencoded) in the output-file.

Part II — The control-file

5. The basic input statements

The control-file contains the main instructions for the program, expressed as *statements* of various kinds. In auto-mode, its name should be an argument of the command that launches the program (Section 35); in API-mode, it must be passed in one of the arguments of the interface subroutine (Section 37.4).

5.1 The required statements

Although the control-file may include quite a few different types of statements (see Section 21), we will start off with a 'minimal' example which will help us discuss the basic statements, possibly in their simplest form; both auto-mode and a single output-file are assumed. Any other type of statement is optional (sometimes the `output`, `style`, and `loop_momentum` statements can be absent too).

```
output = 'q_list' ;
style = 'qgraf.sty' ;
model = 'qed' ;
in = electron[p1], positron[q2] ;
out = photon[q1], photon[q2] ;
loops = 2 ;
loop_momentum = k ;
```

These statements cannot appear in an arbitrary order; with a single exception (the first two can be permuted) their relative ordering is fixed. The generic statement ordering is shown in Section 22.

The `output` statement declares an output-file, which is a file where the generated diagram descriptions should be included, one after another. This statement is not available in API-mode. Unless it is known in advance that the number of diagrams is not too large, a preliminary run, with no output, should be considered — see eg the `count_to` statement (Section 16).

The `style` statement declares a style-file, which should contain the instructions for generating an output-file.

The `model` statement declares the model-file, which should consist in a description of the model for which the diagrams are required.

In the above example, all of the input and output files are in the current directory. Yet, it is possible to declare files in other directories, eg

```
model = 'models/qed' ;
```

or even is some other directory, eg

```
model = '/home/username/models/qed' ;
```

Moreover, three special (sub-)directories can be declared — one for model-files, another for style-files, and yet another directory for output files (Section 7).

In any statement specifying some input or output file, or some default directory, the respective name should be the encoded form of the actual filename used by the operating system. Nevertheless, allowed paths and filenames are subject to the constraints described in Section 4.2 and in Section 7.3.

The `in` and the `out` statements declare the incoming and the outgoing fields (which should be defined in the model-file) as well as the respective momenta (which should be identifiers). In the above example, `p1` is the momentum of the `electron` (flowing inwards) and `q1` the momentum of one of the `photon` fields (flowing outwards). Additional details are presented in Section 10. It should be noted that the diagram sign may depend on the exact sequences in which the incoming fields and outgoing fields are declared (Section 38).

The `loops` statement specifies the number of loops (or cycle-rank) of the diagrams to be generated. An extended form of this statement also exists (Section 11).

The `loop_momentum` statement declares the common prefix of the integration momenta, which must be an identifier. For example, if the control-file declares that identifier as `k` and requests 2-loop diagrams then the two integration momenta will be `k1` and `k2`. This statement is optional when the respective identifier is not needed — eg when no output-file is to be produced, or when the diagrams requested are trees. Conflict with the zero-momentum and the external momenta should be avoided.

5.2 Void statements and superfluous statements (in the control-file)

A *void* statement is a statement that includes no actual data, eg

```
config = ;
style = '' ;
```

Currently, only the following types of (valid) statements may be void.

- `config`
- `in`
- `options`
- `out`

When the control-file is edited 'by hand', statements can easily be converted into annotations and back again.

Void statements are not the only kind of *superfluous* statements. Sometimes it is possible to delete a non-void statement from the control-file without producing any relevant effect. Any non-void superfluous statement is accepted provided its type is usually accepted (note that two types of statements are not available in API-mode Section 21).

5.3 Long filenames (and long paths)

The string encoding described in Section 4.1 can be employed to deal with long 'file-names' (path included). As the filenames declared in the control-file are encoded strings, they can be split into separately encoded components. For example, the statement

```
model = '/home/username/models/qed' ;
```

can also be written as (eg)

```
model = \
  '/home/username/' \
  'models/qed' \
   ;
```

(possibly without statement continuation marks, for now). Therefore, the length of such a filename can easily exceed the maximum allowed line length. As mentioned earlier, there is also an upper bound for the statement length... (not smaller than 500).

6. The `config` statement

The `config` statement allows the specification of diagram unrelated options. If present, it must be the first statement in the control-file. The keywords allowed in this statement will be dubbed *config options* (or configuration options), to distinguish them collectively from the other kinds — ie diagram options and command-line options. Some of the config options (`nolist`, `lf`, and those that define the display-modes) are not available in API-mode.

6.1 The option `nolist`

The config option `nolist` disables creating any output-file when one or more files of that kind are declared, eg

```
config = nolist ;
output = 'd_list' ;
style = 'f0.sty' ;
        ⋮
```

Thus, this option can be used to predict whether some output-file, to be (possibly) generated in a subsequent run — a similar one, just without that option — will be of any use (and with no risk of filling up the disk partition). A *'bounded run'* can be performed with the help of the `count_to` statement (Section 16).

6.2 The option `delete`

Whenever the filename specified in either the `messages` or `output` statements is that of an already existing file *and* the output-directory is defined (Section 7.2), the config option `delete` instructs the program to overwrite *any* such obstructing file. An example follows.

```
config = nolist, delete ;
output_dir = 'tmp_dir/' ;
messages = 'msg.txt' ;
output = 'q_list' ;
        ⋮
```

This option should be used with care.

6.3 The option `lf`

If the operating system is one of those for which a *'newline'* consists of a `line feed` character (ASCII control character `LF`), there is an experimental feature that should speed-up the write operation and which is enabled by the config option `lf`, eg

```
config = lf ;
```

This should lead to a small to moderate (overall) performance increase, if the output-file is large enough. Compatible operating systems include Linux/GNU based systems, as described in Wikipedia's *Newline*[7] page, Section *Representation*. One may always run the program with and without that option (a single test case might be sufficient) and check whether the output-files are identical.

[7] `https://en.wikipedia.org/w/index.php?title=Newline&oldid=863813417`

6.4 The option `noblanks`

The config option `noblanks` instructs the program to discard *blanks* (ie each and every space character) appearing in the names of files and directories (paths included) read from the control-file. For example, this means that the statements

```
config = noblanks ;
output_dir = ' tmp_dir / ' ;
output = ' q_list ' ;
```

should become equivalent to

```
output_dir = 'tmp_dir/' ;
output = 'q_list' ;
```

irrespective of the operating system.

6.5 The display-modes: `noinfo`, `info`, `verbose`

Given the current number of possible warning messages, it may be convenient to have some control over the output displayed (in auto-mode). Three *display-modes*[8] have been created for that purpose, and can be described as follows.

- `noinfo`

  Nothing should be displayed unless an error condition is detected.

- `info`

  The program's name and version numbers, the statements from the control-file, and various numbers of generated diagrams (subtotals for each vertex-degree partition, and also the total number) are displayed in that order; when applicable, the subtotal for each loop order is also displayed. If the diagram generation finishes but there were suppressed warning messages (which would have been shown in `verbose` mode), a warning sign is added to the line that shows the total number of diagrams, eg

  ```
  total =  12343 connected diagrams  (w!)
  ```

- `verbose`

  In addition to the information displayed in `info` mode, the program displays every 'alert' it is able to, as well as a summary of the model consisting of various numbers of propagators, vertices and (when applicable) sectors; as it should be clear, this mode may help in fixing defective input files.

The display-mode can be set explicitly by including one of those three options in the `config` statement, eg

```
config = info ;
```

Otherwise, it is set implicitly: if the config option `nolist` is used then the display-mode is set to `verbose`, else to `info`. If a run-time error is detected, the corresponding error message can be either displayed or saved in the message-file (or both) and then the program stops.

[8] A display-mode corresponds to what is often called a verbosity level.

6.6 The option `flush`

Normally, the program will exit without ascertaining that the operating system makes any open output files available to other processes, in their final form (although the operating system will eventually do so). Using the config option `flush` prevents that kind of short-cut and might therefore be useful (specially) when some of the output-files are big: the program will explicitly ask the operating system to do the above mentioned operation, which may speed-up its completion (probably slightly), and then when QGRAF exits other programs should be able to start reading the output files at once (this is likely similar to executing a `sync` command applied only to those files).

6.7 The option `no_ntnls`

In API-mode, trailing *newlines* are automatically deleted but non-trailing ones are not. While it is possible to rewrite a style-file formerly designed for auto-mode into another more appropriate for API-mode, the config option `no_ntnls` offers a different possibility: it will enforce the deletion of non-trailing newlines in any output-block for the diagram section (only after the output-block has been built in the usual way).

This option can be used in auto-mode too.

6.8 A missing option

When the number of diagrams is ‘big’ (greater than 10^5, say), the output-file will be quite large, obviously, and it might be desirable to split it in several files of a more manageable size if (eg) the diagram processing software can handle them more efficiently than it would handle a single large file. Although the program does not provide a way to perform this kind of operation, that might not be a real problem as there exist tools which, with little effort, can be used for the job. The rest of this section describes a possible method (for Linux/GNU operating systems). The following lines

```
#␣<diagram_counter>xyx
<epilogue>
```

show an excerpt of a conceptual style-file: they represent the last line of the diagram section (an extra, artificial line introduced for the present purpose) and the line that declares the epilogue section. Let us assume that the string xyx does not normally appear in the output-file, ie that it is generated only when that extra line is added (if that is not the case then some other string may have to be used). Now we will rely on the operating system and execute a command of the following kind.

```
csplit --prefix='xx' --digits=3 dlist '/0000xyx/' '{*}'
```

This splits output-file `dlist` into smaller files (the *pieces*), containing the description of 10^4 diagrams each (except for the last piece, which could have a smaller number), as the pattern `0000xyx` appears every time the diagram counter is a multiple of 10 000. In this example the pieces will be named `xx000`, `xx001`, `xx002`, and so on; the prefix `xx` and the length of the suffix (ie the number of digits) may be specified as command-line arguments.

The prefix should be chosen with care, as `csplit` overwrites existing files; it is safer to execute the `csplit` command in a directory containing only (a copy of) the necessary files, of course. If the lines matching the pattern should be deleted, the following additional option can be used.

```
--suppress-matched
```

To keep the number of files within reasonable bounds (let us say that we want at most 100 files, approximately), the number of diagrams per piece should typically be larger than 1% of the total number of diagrams. Hence, for that upper bound, having 10^4 diagrams per piece could be acceptable when the total number of diagrams does not exceed one million.

That process can be iterated without having a very large number of files at any given time. For example, if the output-file contains 10^7 diagrams and one wishes to create pieces with 10^3 diagrams each, one may first create only 100 pieces with 10^5 diagrams each and then split each piece in 100 sub-pieces, but not simultaneously — ie a piece is split, its sub-pieces processed and then deleted, then another piece is split, and so on.

In the near future, hopefully, a direct interface to each diagram amplitude (as generated in real time) should provide a flexible solution to the problem of storing and/or processing the program's output in more diverse ways.

7. The model_dir, output_dir, style_dir and separator statements

A *default directory* is a directory where the program should look for input files of a certain type (either model-files or style-files), or where the files created by the program (message-files and output-files) should be saved. Three new (optional) statements provide a way to define three types of default directories; moreover, to enhance the compatibility of the program with different operating systems, the path (or directory) separator can be defined with the help of a further statement. Recall that the names of files and directories accepted by the program are somewhat restricted (Section 4.2).

7.1 The model_dir and style_dir statements

The following example shows how to set default directories for model-files and for style-files, which will be dubbed the model-directory and the style-directory.

```
config = ;
model_dir = 'models/' ;
style_dir = 'styles/' ;
output = 'q_list' ;
style = 'style1' ;
model = 'model2' ;
        ⋮
```

The program will then prepend (eg) the name of the model-directory (specified in the `model_dir` statement) to the name of the model-file (specified in the `model` statement) to construct the actual filename to be passed on to the operating system; a similar convention applies to style-files. In the above example, the program would ask the operating system to read the following files.

```
styles/style1
models/model2
```

Note the trailing slash in the strings that define default directories, which is there because the program ignores which operating system is being run (see also Section 7.3, however).

7.2 The output_dir statement

The `output_dir` statement sets the *output-directory*, which should be *reserved* to the files produced by the program. In the next example,

```
output_dir = 'output/' ;
messages = 'msg.txt' ;
output = 'q_list' ;
        ⋮
```

the program would try to create the files

```
output/msg.txt
output/q_list
```

and, as usual, would make no attempt to create alternative files if anything went wrong. The full names of the output-file(s) *and* of the message-file are obtained by prepending the string that defines the output-directory to every filename specified in either an `output` or `messages` statement.

There is an additional feature that is not available for the other default directories; namely, it becomes possible for the program to *delete* any file in the output-directory if instructed to create one or more files that already exist. For that to happen, however, the config option `delete` should be used, eg

```
config = delete ;
output_dir = 'output/tmp_dir/' ;
messages = 'msg.txt' ;
output = 'q_list' ;
          ⋮
```

Verifying that the `output_dir` statement is working as expected, before using option `delete`, is strongly recommended.

7.3 The `separator` statement

The slash '`/`' is used as a path (or directory) separator by (eg) Linux/GNU systems, but other operating systems may use a different character. That mark might be definable with the help of the `separator` statement, eg

```
separator = '/' ;
```

A single non-alphanumeric character is expected, though 'encoded' (note that the `noblanks` option does not apply to this statement).

If a path separator is defined explicitly then the names of the default directories need not end with it — if they do not, the program will automatically append one such character when necessary.

On the other hand, if a path separator is not defined explicitly then the names of those directories have to end with a (common) non-alphanumeric character, which will be assumed to be the path separator.

The name of a regular file (or of a symlink pointing to a regular file) should not end with a path separator.

NB: Although `qgraf-4.0.5` accepts only one separator character (namely, the slash), other separators may be added if necessary; a detailed request may be sent in by e-mail after confirming that the website's page '`news, alerts`' has no entry about such a request.

8. The messages statement

The `messages` statement declares the name of a file (dubbed the *message-file*) to be created by the program for saving error and warning messages, eg

```
config = verbose ;
output_dir = 'tmp/' ;
messages = 'msg.txt' ;
output = 'd_list' ;
style = 'f0.sty' ;
        ⋮
```

Since the program stops on any detected error, if an error message is saved then it should be the last message in that file. If empty, the message-file is deleted at the end of the run. Moreover, the symbol `(w!)` mentioned in Section 6.5 will not be displayed if the message-file has been created (whether it is kept or not).

This optional statement is independent of the display-modes and aimed mainly at script based set-ups. Note, though, that messages issued before the message-file has been opened will not be stored in that file. Problems at that stage should occur only rarely, if at all, unless

- the control-file cannot be read or is defective in some way (for example, if it includes invalid statements, or statements that lead to errors), or
- the message-file already exists and cannot be overwritten (see config option `delete`, Section 6.2).

9. Multiple `style` (and `output`) statements

To generate more than one type of output-file in the same run, the control-file should include multiple `output` and `style` statements. In this case, the style-files and output-files should be paired — by alternating the respective statements, eg

```
config = ;
style = 'f1.sty' ;
output = 'out1' ;
style = 'f2.sty' ;
output = 'out2' ;
       ⋮
```

All of those files should be distinct, and there should be at most 7 pairs. It is possible to have zero pairs, although that case is not very useful (still, the number of diagrams may be obtained). Note that 'void' `style` and `output` statements are no longer allowed.

The program tries to open every style-file specified in the control-file, almost simultaneously, and a run-time error occurs if two (or more) of those files are actually the exact same file. For example, that error will occur if the control-file includes the statements

```
style = 'f1.sty' ;
style = 'f2.sty' ;
```

and if `f2.sty` is a symlink to `f1.sty`, since in that case the program would try to open the same file twice.

In API-mode, multiple `style` statements may still appear but no `output` statement is allowed. In this mode there are no output-files, just output-blocks.

NB: For each output-file, the `<statement_loop>` skips the iterations corresponding to those `output` and `style` statements that do not refer to that file.

10. The `in` and `out` statements (extended version)

Occasionally, the external momenta need not be declared and it is possible to state (eg)

```
in = electron, positron ;
out = photon ;
```

If at least one momentum is declared, then all momenta must be declared. Nevertheless, when the output of the program depends one some momentum, internal or external (which very likely means nearly always), the external momenta do have to be declared,[9] eg

```
in = electron[p1], positron[p2] ;
out = photon[q1] ;
```

Should anti-commuting fields be present, some care should be taken with the relative ordering of the external fields declared in those statements — specially if two or more runs are employed to create the required output-files for the same scattering process (Section 38).

The expressions for the momenta of any diagram are seldom unique, in terms of both the external momenta and the integration momenta. Given that the sum of the incoming momenta must be equal to the sum of the outgoing ones, it is always possible to avoid including in such expressions a term that depends on a specific external momentum. That can be achieved by altering the respective declaration in a subtle way. In the previous example, to exclude any term for the momentum `q1`, the following `out` statement should be employed instead.

```
out = photon(q1) ;
```

This type of substitution modifies the output of the style-keywords `<momentum_term>`, `<momentum>`, and their duals, but only in the propagator-loop and the ray-loop. Their output remains unaffected in both the in-loop and the out-loop.

For n-point functions, $n > 2$, such substitutions tend to increase the average number of terms in the expressions for the internal momenta, but for $n \leq 2$ that is not so.

For $n = 2$, it becomes possible to have the internal momenta depend on only one (fixed) external momentum (besides the integration momenta, of course).

For $n = 1$, the question does not arise except in the ray-loop, and then only for the unique (vertex,ray) combination that corresponds to the leg of the diagram; in this case, that momentum is replaced by the zero-momentum. No internal momentum depends on the external one (which is null by momentum conservation, anyway).

[9] There are no longer default momenta, as relying on their existence could shorten the path to trouble.

11. The `loops` statement (extended form)

From the very start, the `loops` statement allowed users to specify exactly one value (the *number of loops*, or *cycle-rank*), eg

```
loops = 2 ;
```

It is now possible to instruct the program to generate in a single run all the diagrams for a number of consecutive cycle-ranks, eg

```
loops = 1 to 4 ;
```

or even

```
loops = 4 thru 1 ;
```

These two statements are not fully equivalent, even though the keywords `thru` and `to` can be exchanged with one another. The program starts with the first (ie leftmost) specified value, and then progresses sequentially towards the second value by adding or subtracting 1 in each iteration, as appropriate. The diagram constraints specified by other statements apply to every loop order, obviously.

Usually, the propagators do not contribute directly to the order of perturbation theory (they contribute with a null weight, let us say) whilst a vertex of degree d (≥ 3) contributes with weight $(d/2)-1$, which is positive. Actually, it may be more convenient to consider the expansion of the amplitude in some coupling constant g (or in some other appropriate factor) rather than on the number of loops, so that the weights are integer (in which case they can be accepted by `vsum` statements). For example, at tree-level, in a simple gauge theory, cubic vertices are proportional to the coupling constant g and quartic vertices are proportional to g^2 (and propagators do not depend on g).

Nonetheless, there are cases in which the number of loops given as input does not represent the order of perturbation theory (when the vertices and/or propagators represent some higher order corrections instead of the tree-level interactions and propagators). It will then be necessary to deal with propagators and/or vertices with unusual weights. Often, such cases can be addressed by performing multiple runs (for different values of the cycle-rank), with specially adjusted parameters for each run — including the numerical arguments for the `vsum` and/or `psum` statements. Then, the extended `loops` statement might be able to reduce the number of required runs.

There is a marked difference between dealing with usual or with unusual weights, however. With the usual (implicit) weights, a single run with an extended `loops` statement provides the required diagrams for all of the corresponding orders of perturbation theory; in the other case, and assuming that the order of perturbation theory can be defined in terms of the number of loops and of a *single* (explicit) weight, each run provides the required diagrams for just one such order.

NB: The display-output has been modified, so that the cycle-rank is displayed in the diagram generation phase too. The current potential of this extended statement is still a bit limited, but that should change whenever some other improvements are introduced.

12. The `options` statement

The `options` statement can be used to specify a number of (mostly) topological properties that the generated diagrams should have. If no such keywords are stated then the program will try to generate all the connected diagrams satisfying the constraints imposed by the existing statements. To make QGRAF discard certain types of diagrams (such as diagrams with tadpoles, 1-particle reducible diagrams, etc) one simply has to include the appropriate keywords, separated by commas, eg

```
options = nobridge, noselfloop ;
```

12.1 The keywords for the `options` statement

The current list of *topological* options for the `options` statement follows — each 'main' option on the left-hand side, 'complementary' option on the right-hand side, forming pairs of dual options. These options can be used to impose various types of restrictions on the unlabelled topologies of the generated diagrams, irrespective of the fields involved.

`bipart`	`nonbipart`
`cycli`	`cyclr`
`nobridge`	`bridge`
`nodiloop`	`diloop`
`noedge`	`edge`
`noparallel`	`parallel`
`norbridge`	`rbridge`
`nosbridge`	`sbridge`
`noselfloop`	`selfloop`
`nosigma`	`sigma`
`nosnail`	`snail`
`notadpole`	`tadpole`
`onepi`	`onepr`
`onevi`	`onevr`
`onshell`	`offshell`
`onshellx`	`offshellx`
`simple`	`notsimple`

In (classical) graph theory, there are certain graph transformations that involve the *deletion* of nodes and/or edges; any edge is indivisible and must be automatically removed if any of its endnodes is deleted. A different option, which seems more natural in a QFT context, consists in defining an edge as a pair of linked *half-edges*. Moreover, any vertex includes a set of half-edges, each of which may or may not be linked to some other half-edge. One will then speak of *splitting* or *cutting* edges (into the constituent half-edges), thereby preserving the vertices involved. Also, *deleting a vertex* will mean splitting every edge incident to that vertex first, and then deleting the full (isolated) vertex.

There are also a few other options (of different kinds), shown next.

```
floop               notfloop
new_elinks
new_loops
new_partition
new_topology
topol
```

12.2 The main options

The remainder of this section contains a description of all those options, in alphabetical order — but, typically, only for the one of the options of each dual pair. When a dual exists, each option rejects what its dual selects, and conversely; hence there is no need for repeating this part of the definition over and over.

• `bipart`

This option selects those diagrams whose topology is a bipartite graph.

A bipartite graph is a graph whose node-set can be partitioned into two subsets A and B in such a way that every edge (if any exists) joins a node $u \in A$ to a node $v \in B$. That is the same as selecting the diagrams with no circuit (or *loop*) of odd length; in particular, the diagrams must have no self-loops (nor triangles, pentagons, and so forth). Every tree is clearly bipartite.

The dual option (`nonbipart`) requests at least one odd cycle.

• `cycli`

This option selects those diagrams that have at most one *'cycle-block'*.

Given the usual types of Feynman rules in momentum space, the evaluation of some diagrams involves a factorizable integration, ie decomposable into a product of two or more independent integrations (for 2-loop and higher order diagrams, obviously). Even if the complete integrand does not have such a property it might be possible to decompose it into a sum of factorizable expressions, if the diagram topology is right.

A circuit (or simple cycle, which physicists may call a loop) of a graph G is a non-empty set of edges that, together with their endnodes, define a connected subgraph of G in which every node has degree 2. It follows from this definition that a circuit cannot be decomposed into two or more circuits. Recall that an edge of a graph is either a bridge (if it does not belong to any circuit) or a nonbridge.

Definition: a cycle-block (or cycle-component) of a graph G is a non-empty set S of nonbridges such that

◦ for every circuit C of G, either $C \subseteq S$ or $C \cap S = \emptyset$;

◦ given any two edges $e_1, e_2 \in S$, there is a circuit containing e_1 and e_2.

Circuits of length 1 and 2 are allowed: the former case corresponds to self-loops, the latter to pairs of parallel edges (self-loops excluded) as in diagrams D_1 and D_3 (Fig. 2).

There is a simple interpretation of cycle-blocks in terms of sub-diagrams. Let D be an l-loop, non-tree diagram, and P_c the set of its nonbridge propagators; also, let $k_1, k_2 \ldots k_l$ be the independent integration momenta, and ignore the external momenta (eg set them to zero). Then each cycle-block is a minimal (ie non-decomposable into two or more sets of the same kind) non-empty subset S of P_c such that the two sets of momenta flowing through the propagators in S and $\bar{S} = P_c \setminus S$, respectively, can be parametrized independently. The case where $S = P_c$ and $\bar{S}$ is empty is implicitly allowed, as there may exist a single cycle-component.

The diagrams selected by option `cycli` (which stands for *cycle-irreducible*) are those that have a non-factorizable cycle space, ie that have at most one cycle-block. Since `cycli` ignores bridge-type propagators, it is the following combination

```
        options = cycli, onepi ;
```

which selects sets of diagrams even more 'primitive' than those selected by option `onepi` only.

Any self-loop is a cycle-block. Diagram D_1 (Fig. 2) has two self-loops and thus two cycle-blocks (hence it is rejected by `cycli`); denoting the momenta of propagators 1 and 2 (as marked) by k_1 and k_2, one may readily see that (a) those two momenta are independent, and (b) there is no propagator whose momentum has to be expressed as a linear combination in which the coefficients of k_1 and k_2 are both nonzero.

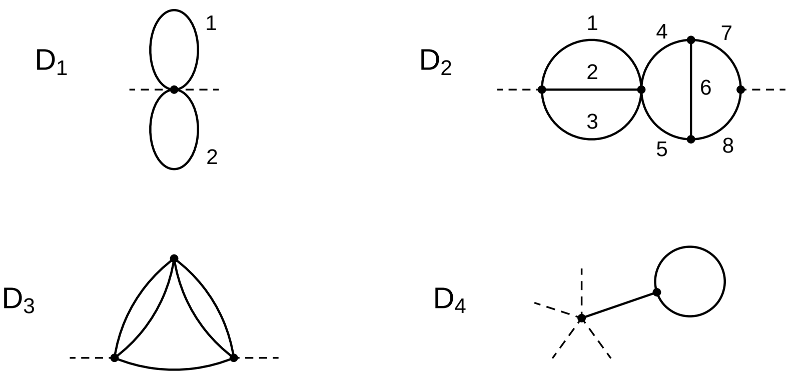

Fig. 2. Illustrating option `cycli`.

Diagram D_2 has also two cycle-components, defined by the subsets of propagators $\{1, 2, 3\}$ and $\{4, 5, 6, 7, 8\}$. Excluding the contributions of the external momenta, here it is possible to express eg k_3 in terms of (k_1, k_2), and (k_5, k_6, k_8) in terms of (k_4, k_7). Since one may do so for each of the above (disjoint) subsets, D_2 is rejected too. That type of partitioning is not possible for diagram D_3, which is selected. Any tree diagram and 1-loop diagram (eg D_4) is selected by `cycli`.

• `floop`

The diagrams rejected by option `floop` are those that have at least one circuit (ie loop) of odd length for which every propagator is that of an anti-commuting field. In `qgraf-3.2` and later versions the use of `floop` requires that

- ∘ there is at least one anti-commuting field;
- ∘ every vertex has either 0 or 2 anti-commuting fields.

- `new_elinks`

  This option selects only one diagram per labelled topology — more precisely, it selects those diagrams for which `<new_elinks>` produces a '`1`'.

  Option `new_elinks` may simplify the task of determining the distinct labelled topologies of the output diagrams. It is incompatible with options `new_loops`, `new_partition` and `new_topology`.

- `new_loops`

  This option selects only one diagram per cycle-rank (ie number of loops) — ie it selects the diagrams for which `<new_loops>` produces a '`1`'.

  It is incompatible with options `new_elinks`, `new_partition` and `new_topology`.

- `new_partition`

  This option selects only one diagram per vertex-degree partition — that is, it selects the diagrams for which `<new_partition>` produces a '`1`'.

  It is incompatible with options `new_elinks`, `new_loops` and `new_topology`.

- `new_topology`

  This option selects only one diagram per unlabelled topology — that is, it selects the diagrams for which `<new_topology>` produces a '`1`'.

  Option `new_topology` may simplify the task of determining the distinct unlabelled topologies of the output diagrams. It is incompatible with options `new_elinks`, `new_loops` and `new_partition`.

- `nobridge`

  This option rejects topologies with bridges (thus, `nobridge` is equivalent to `onepi`).

- `nodiloop`

  This option rejects topologies with circuits of length 2 (that is, any two distinct vertices may be joined by at most one propagator).

- `noedge`

  This option excludes diagrams with edges (or propagators). Thus, it selects only those diagrams that consist of a vertex whose legs coincide with the external fields of the input process (it could be more than one diagram, since duplicate vertices are allowed).

- `noparallel`

  This option excludes diagrams with parallel edges (ie no di-loops, and each node can have at most one self-loop).

• `norbridge`

This option rejects diagrams with *regular* bridges.

A bridge is regular if its splitting divides the (connected) diagram into two sub-diagrams, both of which include at least one external field (that is the type of bridge which is *not* associated with tadpoles). Option `norbridge` can be used to generate the diagrams for those correlations functions analogous to 1-particle irreducible correlations functions, but possibly with tadpoles (ie with singular bridges).

• `nosbridge`

This option excludes diagrams with *singular* bridges.

A bridge is singular if its splitting breaks the (connected) diagram into two sub-diagrams such that (at least) one of the sub-diagrams does not include any external field. In other words, a singular bridge is a non-regular bridge, and a regular bridge is a non-singular bridge.

• `noselfloop`

This option excludes diagrams with self-loops.

A self-loop is an edge whose endnodes coincide (ie which forms a circuit of length 1).

• `nosigma`

This option excludes diagrams with *'self-energy'* insertions (ie sub-diagrams that are 2-point functions), anywhere. Since the criterion is topological, it can exclude diagrams that are not higher-order corrections of a tree-level propagator — that is, diagrams with 'mixed' propagators are also rejected.

• `nosnail`

This option is an extension of `notadpole` that also rejects diagrams which contain what one might call *collapsed tadpoles* (or *tadpoles without tail*).

• `notadpole`

This option is equivalent to option `nosbridge`.

• `onepi`

This option (which is equivalent to option `nobridge`) excludes diagrams with bridges, usually dubbed *1-particle reducible* diagrams.

A 1-particle irreducible diagram is a connected diagram that cannot be disconnected by cutting any single propagator.

• `onevi`

This option selects those diagrams whose topology has no articulation point, or cut vertex (in the graph theoretical lingo). This means those diagrams that remain connected upon the removal of any single vertex. By definition, the empty graph is connected; consequently, diagrams discarded by option `onevi` must have at least three vertices. The dual option is `onevr` — `onevi` means *1-vertex irreducible*, and `onevr` means *1-vertex reducible*.

In the case of 1-particle irreducible diagrams there is some overlap between options `cycli` and `onevi`, but they do not coincide. If both bridges and self-loops are excluded, however, then they become identical.

In Fig. 3, T_1 and T_2 are 1-vertex reducible: deleting any vertex that does not link to an external field generates two disjoint components. Topology T_3 is 1-vertex irreducible as it has too few vertices to be reducible. Some of the previous figures provide useful examples too: diagram D_3 in Fig. 6, and diagram D_2 in Fig. 2 are 1-vertex reducible.

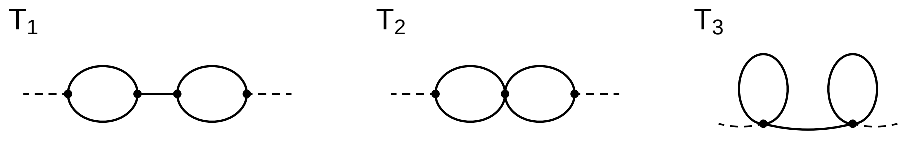

Fig. 3. Illustrating option `onevi`.

There is a class of topologies that are both 1-particle reducible and 1-vertex irreducible, for which T_3 (Fig. 3) may be seen as a kind of prototype: they have exactly two nodes (u and v, say), joined by a single edge; u and v may have multiple self-loops (depending on the model); the external fields connect to u and/or v, obviously.

• `onshell`

This option rejects diagrams with *'self-energy'* insertions (ie 2-point functions) on the external lines — where a propagator with a momentum equal to that of an external field is present. As the criterion is topological, those 2-point functions need not be propagators of the classical Lagrangian). See also options `nosigma`, `onshellx`, and the `plink` statement.

• `onshellx`

Option `onshell` discards diagrams that have a propagator whose splitting generates two separate diagrams, provided one of those is a 2-point diagram. In that case there will be a bridge-type propagator whose momentum equals one of the external momenta (as in diagrams D_2 and D_6 (Fig. 4), where the bridge is labelled with the letter b and the corresponding external line with the letter e).

Occasionally, a more extensive elimination may be useful. Option `onshellx` also rejects diagrams for which the above mentioned bridge is absent, as if it had contracted to a single point and its two end-vertices had merged (compare eg D_1 and D_2). Diagrams rejected by `onshellx`, but not by `onshell`, will have at least one vertex of degree $d \geq 4$. As those diagrams can be generated by using the combination

```
options = offshellx, onshell ;
```

it is possible to do a basic cross-check in the case where they are thought to evaluate to zero.

Diagram D_1 is rejected by option `onshellx` (but not by `onshell`), whether or not there exists (in the same model) a similar diagram D_2 with the same fields except for the bridge *b*. Diagram D_2 is discarded by either option, obviously.

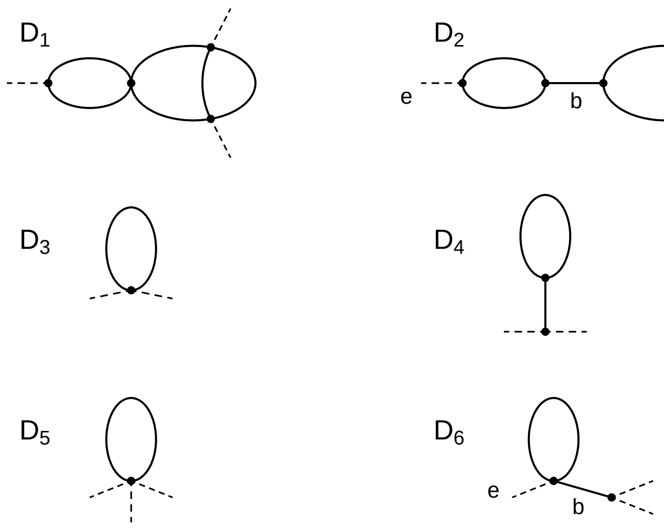

Fig. 4. Illustrating option `onshellx`.

Diagram D_3 is not rejected by `onshellx`, unlike D_5. Although D_3 can be obtained by contracting the bridge in D_4, that bridge does not isolate a single leg (and no suitable diagram exists), whilst D_5 can be obtained from D_6, which has a suitable form. If options `onshellx` and `nosnail` are used simultaneously then all six diagrams will be rejected.

Regarding cross-checks that have been performed so far, the numbers obtained for the combination

```
options = offshellx, nosnail ;
```

agree with several numbers from L. Dixon (private communication) and S. Badger for 1-loop and 2-loop diagrams, respectively. Additionally, a second algorithm has been developed for `onshellx`, and an agreement with the original algorithm was found.

• `simple` (obsolescent)

This option excludes topologies with self-loops and di-loops. The following equivalence applies.

`simple` $\Leftrightarrow$ `noselfloop, nodiloop`

• `topol` (obsolescent)

This option discards diagrams whose (unlabelled) topology is equal to that of an earlier generated diagram (to be used with a single neutral field, only).

Option `topol` can be replaced by `new_topology`, which is more general; it will not be available in the next stable version.

13. The zero_momentum statement

The zero_momentum statement can simplify the implementation of a proper ‘space-time index algebra’, with every 4-momentum (or perhaps D-momentum) carrying its own space-time index. Moreover, concerning the symbolic manipulation of the amplitudes, the actual zeroness of that momentum can then be ignored until the most convenient moment.

This type of statement simply declares an identifier, eg

```
zero_momentum = k0 ;
```

that becomes the redefined null momentum (which had been set to ‘0’ by default). This identifier (here, k0) has to be distinct from the other basic momenta (external momenta and integration momenta). The consequence of that redefinition is this: when a null (internal) momentum is found and the respective expression must be produced, the program outputs that identifier instead of the usual ‘0’.

The natural use of this statement is associated with the <momentum_loop> construct (Section 32.9), although the above mentioned substitution applies also to the output produced by <momentum>. In an output-file, one can then have an expression like k0_{mu4} instead of 0_{mu4}.

14. The `partition` statement

Unless the `noinfo` option is enabled, the graphical output includes the *vertex-degree partitions* that are compatible with all of the following inputs: the model, the scattering process, and the order of perturbation theory. In general, this compatibility is not sufficient to ensure the existence of corresponding diagrams in the input model — it just means that the vertex degrees and their respective multiplicities satisfy a simple arithmetical relation which ensures that there is at least one unlabelled topology with that vertex-degree partition (which in *some* models with the same set of vertex degrees will be the topology of some diagram).

The `partition` statement restricts the diagram generation to some of those partitions. For example, if a model has cubic and quartic vertices only, the statement

```
partition = 3^2 4^1 ;
```

requests that the diagrams should have precisely two vertices of degree 3 and one vertex of degree 4. Suppose now that the set of vertex degrees of the input model is $\{3, 4, 5, 6\}$. Then, since any missing term is ignored, the above statement may select more than one partition; in fact, it selects every compatible partition of the form $3^2\ 4^1\ 5^a\ 6^b$, if any, where a and b are 'free' (ie not constrained by that statement). Similarly, the statement

```
partition = 3^0 ;
```

selects those partitions that exclude cubic vertices. A 'term' `n^k` should not appear in the `partition` statement unless the input model contains some interaction vertex of degree `n`. To pick a single partition it may be necessary to specify nearly all (or even all) of the vertex degrees and their multiplicities.

In the above examples each term sets the exact multiplicity `k` of some degree `n`, but it is also possible to impose inequalities rather than equalities. For example, the statement

```
partition = 3^(1+) 4^1 5^(1-) ;
```

requests at least one cubic vertex, exactly one quartic vertex, and at most one quintic vertex. As the next example shows, some `partition` statements are trivial.

```
partition = 3^(0+) ;
```

When the control-file includes a `partition` statement the numbers of diagrams for excluded partitions are not displayed (nor computed, of course), although the partitions themselves are displayed. The following graphical output matches the first of the above statements.

```
 -    4^2
3^2   4^1      ...      10
3^4    -

     total =  10 connected diagrams
```

It should be fairly obvious that the `partition` statement can be employed as a crude method of 'parallelizing' the program (that is, of dividing some diagram generation task, unequally, among a few CPU-cores or a few computers).

15. The `index_offset` statement

When combining the output-files of two or more runs into a single file, it may be useful not to have the first diagram of every output-file being assigned the diagram index `1`. The `index_offset` statement instructs the program to add a non-negative integer to the default diagram index, eg

```
index_offset = 1071 ;
```

The following examples show possible applications of the `index_offset` statement (more complex examples can be devised, of course). Let us suppose in the first place that the diagram selection criteria involve not a conjunction like

```
true = A ;
true = B ;
```

where `A` and `B` represent valid expressions, but some other logical connective, eg

$$\texttt{(true = A)} \vee \texttt{(true = B)}.$$

Although this type of statement is not accepted, there is a way out: that inclusive disjunction can be split into three mutually exclusive cases that can be run separately, namely (case 1)

```
true = A ;
false = B ;
```

then (case 2)

```
false = A ;
true = B ;
```

and finally (case 3)

```
true = A ;
true = B ;
```

Similarly, the exclusive disjunction can be divided in two non-overlapping cases, the equivalence `A` ⇔ `B` in two cases also, and so on.

As a second example, suppose that one wants to generate a set of diagrams satisfying the following two conditions: (*i*) the selection criteria can be expressed as a conjunction of valid statements (hence there is in principle no need to split the run into multiple runs), but (for efficiency reasons, say) (*ii*) the set of diagrams should be partitioned into two subsets, each of which to be processed in a distinct way (or on different computers). Then, if the partitioning can be expressed by a single valid statement (`A`, say), it is as if there was a simple exclusive disjunction with only two cases — namely (case 1)

```
true = A ;
```

and (case 2)

```
false = A ;
```

16. The `count_to` statement

The `count_to` statement, which is not available in API-mode, sets a limit on the number of diagrams to be generated by the program. More precisely, the program stops as soon as one the following targets is reached:

- no further diagrams can be found;
- the number of generated diagrams becomes equal to the positive integer declared in the `count_to` statement.

This statement automatically prevents the creation of any output-file (or output-block), as it is intended solely as a means of obtaining some preliminary information about the size of the set of diagrams to be generated (or not). If no other information is available at the time, the exact number of diagrams will not be known at the end of the run unless it is strictly smaller than the value declared in the statement. For example, if the statement

```
count_to = 46340 ;
```

is included in the control-file, the program will generate at most 46 340 diagrams. If there are fewer diagrams, the total number reported should be the correct number, otherwise merely a lower bound. In the latter case, this uncertainty will be reflected in the final part of the display-output by the addition of a prefix to the string `total`, eg

```
min_total =  46340 connected diagrams
```

Also in that case, the diagram numbers displayed for the last subtotal and for the last vertex degree partition are also lower bounds; naturally, not every (valid) partition and cycle-rank is necessarily displayed.

17. The `elink` statement

The `elink` statement provides a way to restrict the configuration of the external fields — that is, diagrams can be selected or rejected depending on whether or not some external fields are attached to the same vertex or set of vertices. Some constraints of this type, to be dubbed external linking conditions, can be simulated by constructing an extended model — with new `external` fields and new vertices — but that approach can be exacting.

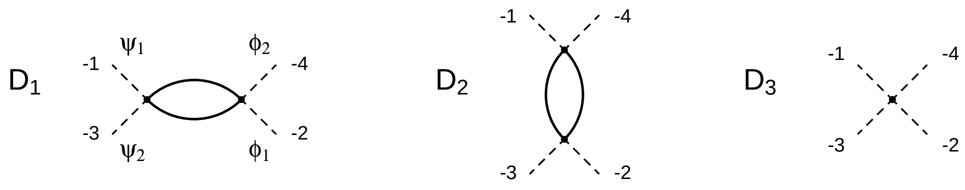

Fig. 5. Illustrating the `elink` statement.

Let us take a look at some examples, assuming the generic scattering process

$$\psi_1\ \psi_2\ \rightarrow\ \phi_1\ \phi_2 .$$

The field-indices of those external fields are, from left to right, equal to -1, -3, -2, -4. The statement

```
true = elink[ -1, -3, incl, 1, 1 ];
```

selects those diagrams in which ψ_1 and ψ_2 link up with the same vertex, *inclusively* — meaning that the other external fields can be attached to any vertex, as far as this statement is concerned. For example (see Fig. 5), diagrams like D_1 and D_3 are validated, and D_2 rejected. This other statement

```
true = elink[ -1, -3, excl, 1, 2 ] ;
```

selects those diagrams in which ψ_1 and ψ_2 link up (in total) with either one or two vertices, this time *exclusively* — ie ψ_1 and ψ_2 may or may not be attached to the same vertex, but in either case the other external fields (ϕ_1 and ϕ_2) should link up with *other* vertices. This condition rejects diagrams D_2 and D_3, for instance.

A generic `elink` statement includes three types of arguments. Each argument of the first type should be the field-index of an external field, and therefore a negative integer (no repetitions are allowed). Then comes a non-numerical argument, either `excl` or `incl`, to specify whether the linking condition is exclusive or inclusive. The third set of arguments consists of two positive integers (a and b, say) which specify the range for the number of vertices involved in the linking condition — that is, for the number of vertices that link up with at least one external field whose field-index is an argument of the `elink` statement. If m denotes the number of arguments of the first type and n the number of legs, then the inequalities

$$1 \le a \le b \le m \le n, \qquad n \ge 2,$$

should hold, otherwise an error will occur. Here are some more examples: the statement

```
false = elink[ -1, excl, 1, 1 ];
```

requests ψ_1 to link up with some other external field(s), whilst the statement

```
false = elink[ -1, -2, incl, 1, 1 ];
```

requests ψ_1 and ϕ_1 not to link up with the same vertex (a condition which diagram D_3 fails to satisfy). For the above mentioned process, the next statement requests that no two external fields link up with the same vertex (actually, in this example `excl` would do as well).

```
true = elink[ -1, -2, -3, -4, incl, 4, 4 ];
```

Then, in the special case of 1-loop diagrams, that statement selects 'squares', or 'boxes'. That possibility can be generalized to select diagrams with a k-cycle ($k \leq n$) from sets of n-leg 1-loop diagrams, should they exist — eg a triangle, a square, a pentagon, and so on (even a 1-cycle and a 2-cycle). In this case (with the above convention) one should have an inclusive statement with $m = n$ and $a = b = k$.

Some `elink` statements are trivial, eg (still in the case of the above mentioned process)

```
true = elink[ -1, incl, 1, 1 ];
true = elink[ -1, -3, incl, 1, 2 ];
true = elink[ -1, -2, -3, -4, excl, 1, 4 ];
```

Their negation rejects every diagram, of course.

18. The `plink` statement

This statement addresses the case where the selection criterion consists in either the absence or existence of a bridge-type propagator with a specific nonzero momentum. The ability to either disallow or require the existence of such propagators may help select eg diagrams whose amplitude has some type of resonance, or tree diagrams contributing to the s-, t- and u-channels; in the case of non-tree diagrams it also provides a way to have some external fields 'on-shell' and others 'off-shell' (as a kind of selective `onshell` option).

If a connected diagram D has a regular bridge then splitting that edge will produce two connected sub-diagrams. Let X_1 and X_2 be the non-empty subsets containing the external fields of D in each of those sub-diagrams; also, let $\mathbf{p}_i$ and $\mathbf{q}_j$ denote the incoming and the outgoing momenta. The momentum $\mathbf{P}$ flowing through that bridge can be then written as

$$\mathbf{P} = \sum_i a_i \mathbf{p}_i - \sum_j b_j \mathbf{q}_j$$

with $a_i, b_j \in \{0, 1\}$; the global sign will be ignored (in any case, the relative signs of the coefficients in the above relation are fixed). As the external momenta satisfy the momentum conservation relation $\sum_i \mathbf{p}_i = \sum_j \mathbf{q}_j$, it is clear that $\mathbf{P}$ can be expressed as a linear combination of the external momenta in two different ways.

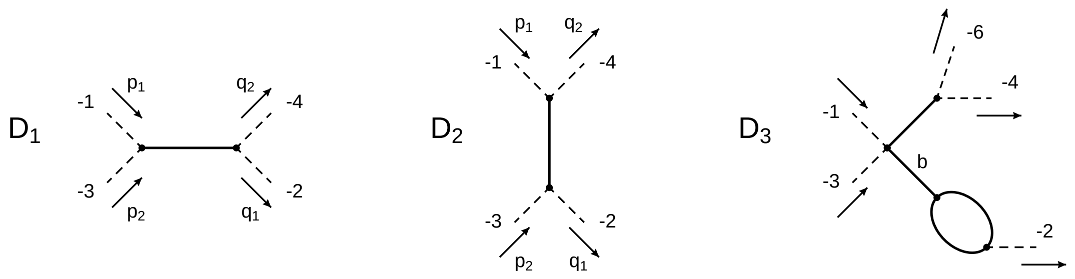

Fig. 6. Illustrating the `plink` statement.

The arguments of the `plink` statement should be the field-indices of the external fields in either X_1 or X_2. The number of arguments cannot be equal to zero, nor equal to the number of legs, as $\mathbf{P}$ would then be null. Let us take a look at some examples: the statement

```
true = plink[ -1, -3 ];
```

selects those diagrams that have at least one propagator with momentum $\mathbf{p}_1 + \mathbf{p}_2$ (or $\mathbf{q}_1 + \mathbf{q}_2$), like diagram D_1 (Fig. 6); diagram D_2, which has a propagator with momentum $\mathbf{p}_1 - \mathbf{q}_2$ (or $\mathbf{p}_2 - \mathbf{q}_1$), would be selected by either of the statements

```
true = plink[ -1, -4 ];
true = plink[ -3, -2 ];
```

The `plink` statement can also be used to set only part of the external fields *'on-shell'*. For instance, the statement

```
false = plink[ -2 ];
```

rejects any diagram with a bridge separating the external field with index -2 from every other external field, such as diagram D_3 (which has one such bridge, labelled with a 'b').

19. The `psum` and `vsum` statements

The ability to define functions (ie parameters associated to fields, propagators, and vertices) in the model-file automatically suggests a few types of diagram selection. The `psum` and the `vsum` statements make it possible to impose some numerical constraints involving integer functions (either p-functions or v-functions).

19.1 The `vsum` statement

The existence of a mechanism to (eg) restrict the powers of coupling constants seems particularly relevant, specially in models with two or more independent coupling constants — partial radiative corrections based on subsets of diagrams defined by such conditions have been routinely considered in the particle physics literature.

Let `g_power` be a v-function mapping each vertex to an integer equal to the power of a certain coupling constant g in the Feynman rule for that vertex. To restrict the power of g in the diagram amplitude, one may write eg

```
true = vsum[ g_power, 4, 4 ] ;
```

In general, the first argument of `vsum` is a v-function and the other two are similar to the corresponding arguments of `iprop`, `bridge`, ..., although they can be negative. Since the function values have to be integer, definitions like

```
g_power = '1/2'
g_power = ' +2'
```

will not be accepted. In principle, it is possible to convert a constraint involving rationals into another constraint depending on integers only; in practice, such integers should not be too large (there should be no problem if their absolute values do not exceed 10^3, say).

Obviously, this statement can be used for purposes other than selecting the powers of coupling constants. For example, let `one` be a v-function that maps every vertex to '`1`'; if `one` is used as the first argument of a `vsum` statement then the number of vertices in each diagram will be restricted.

Here is another example: let V_1 be a subset of the interaction vertices, and `vbinary` a v-function that maps vertices in V_1 to '`1`' and vertices not in V_1 to '`0`'. Then, the statement

```
false = vsum[ vbinary, 0, 0 ];
```

selects diagrams containing at least one vertex in V_1. In this case, defining *submodels* should be considered as an alternative (at least if it makes sense to define such a model).

19.2 The `psum` statement

There is an analogous statement for propagators. For instance,

```
true = psum[ pweight, -1, 1 ] ;
```

restricts the sum of the values of the p-function `pweight` taken over the diagram propagators.

Still, there is a nontrivial difference between the `psum` and `vsum` statements, apart from the fact that they work with distinct types of functions. Given that QGRAF does not produce tree-level 2-point functions, any generated diagram has at least one vertex; therefore, any implicit summation for `vsum` includes at least one term. Nevertheless, there exist tree-level diagrams without propagators — namely, the diagrams that consist of the tree-level vertices of the model, which we may call *stars*. There are also tree-level scattering processes with diagrams of 'mixed-type', ie some diagrams are stars while others contain one or more propagators.

Conventionally, a null term is assigned for the stars, but the user should check whether that is indeed the desired action and, if not, make the necessary adjustments. It might be necessary to generate the stars in a different run (if they are required, but are excluded by some `psum` statement which must be present to deal with the non-stars), or else the stars may have to be excluded (eg) by some additional statement (if stars are to be discarded, but are allowed by the current statements).

19.3 Additional comments

Asymptotically, ie for a large number of vertices (`vsum`) or propagators (`psum`), these 'filters' do not (moreover, cannot[10]) have efficient implementations if they are general enough. For some particular cases, though, some speed-ups could be implemented (note that there has been a relatively small improvement with `qgraf-3.5`).

Update: `qgraf-3.6` brings additional efficiency gains for both `vsum` and `psum`; apart from that, in some cases the impact of this (in)efficiency issue can be lessened by defining submodels (or with the help of the `partition` statement).

[10] At least assuming the widely held belief that the algorithmic complexity classes P and NP do not coincide (ie $P \neq NP$).

20. Other statements (additional filters)

Some other constraints can be imposed on the propagators and the edges with the help of a further set of five statements. These optional statements should be the very last ones to appear in the control-file, and they are of the form

```
<logical> = <operator> [ <arg_1>, <arg_2>, ... <arg_k> ] ;
```

where `<logical>` should be replaced by either `true` or `false`, and `<operator>` by one of the following identifiers.

```
bridge
chord
iprop
rbridge
sbridge
```

The number of arguments (ie k) should be at least 2, and the last two arguments should be non-negative integers (which are subject to the bounds stated in Section 4.3) such that the last argument should not be smaller than the other. The remaining arguments (if any) should be fields.

For example, to restrict the number of propagators of a certain field `phi`, a statement of the following form can be used.

```
true = iprop[ phi, 1, 3 ] ;
```

This statement selects diagrams for which the number of `phi` propagators is at least 1 and at most 3. If `true` is replaced by `false` then the diagrams selected are the ones for which the number of `phi` propagators is either less than 1 or greater than 3. It should not matter whether a propagator is represented by the particle or the anti-particle.

On other occasions one might be interested in propagators with certain topological properties. The 'filters' `chord` and `bridge` restrict the number of propagators that belong (respectively, do not belong) to a *circuit* (or *'loop'*); thus, 'chord' constrains non-bridges and the respective propagators. For example, the statement

```
true = chord[ photon, 0, 0 ] ;
```

requires that there is at least one `photon` propagator in some circuit. All of these operators may have several fields as arguments, for example

```
true = bridge[ photon, electron, 2, 2 ] ;
```

in which case the sum of `photon` and `electron` bridges is constrained, or even no field at all as in

```
true = chord[ 1, 2 ] ;
```

where the total number of chords is restricted.

The statements involving the operators `rbridge` and `sbridge` are similar, and constrain the propagators that sit (respectively) on a regular bridge and on a singular bridge.

NB: A propagator sits on a bridge if and only if its momentum does not depend on the integration momenta; the bridge is singular if its momentum is identically null, irrespective of the external momenta (as in a tadpole with a 'tail'), and is otherwise regular.

It may be observed that the above types of statements may obviate the need for the optional keywords `notadpole` and `external` in the model-file. In fact, statements like the following produce (respectively) the same effect.

```
true = sbridge[ photon, 0, 0 ] ;
true = iprop[ Phi, 0, 0 ] ;
```

The model-file is intended as something fairly permanent, not as something one should change only temporarily for a single calculation, specially if the same result can be achieved with the control-file. Both possibilities exist, however.

21. Statements: required and optional

The `model_dir`, `output_dir`, `style_dir`, `separator`, `zero_momentum` and `count_to` statements are the latest additions. Below, statements marked with an open circle are not available in API-mode. The current, strictly required statements follow, in alphabetical order (these statements should appear exactly once).

- `in`
- `loops`
- `model`
- `out`

Still, the next three statements are typically required too (excepting the `output` statement in API-mode). The `loop_momentum` statement is required if some output that depends on the internal momenta is to be produced. The `output` and the `style` statements, which may appear more than once, are needed to create an output-file; in API-mode, the `style` statement is required if some output-blocks are to be constructed.

- ● `loop_momentum`
- ○ `output`
- ● `style`

The set of optional statements that can appear at most once includes the following.

- ● `config`
- ○ `count_to`
- ● `index_offset`
- ● `messages`
- ● `model_dir`
- ● `options`
- ● `output_dir`
- ● `partition`
- ● `separator`
- ● `style_dir`
- ● `zero_momentum`

Additionally, there exist the optional statements that involve one of the 'filters' listed below, and which may appear more than once.

- `bridge`
- `chord`
- `elink`
- `iprop`
- `plink`
- `psum`
- `rbridge`
- `sbridge`
- `vsum`

22. An example of a modern control-file

```
##   the config options

  config = delete, noblanks ;

##   the path-separator

  separator = '/' ;

##   the default directories
##    (in any relative order)

  output_dir = 'tmp_dir/' ;
  style_dir = 'styles/' ;
  model_dir = 'models/' ;

##   the message-file

  messages = 'msg.txt' ;

##   the style-file(s) and (in auto-mode) the output-file(s)
##    (alternating statements if more than one pair exists);
##   no output-file(s) in api-mode

  style = 'f1.sty' ;
  output = 'q_list.1' ;

##   the main required statements
##    (in the predefined order)

#  model = qed1 // 'qedx' ;

  model = 'qed' ;
  in = mu_minus[p1], mu_plus[p2] ;
  out = photon[q1] ;
  loops = 4 ;
```

(continues on the next page)

```
##   other statements that may appear just once
##    (only the next 6 types, in any relative order,
##     but no count_to statement in api-mode)

  loop_momentum = k ;

  zero_momentum = k0 ;

  options = onepi, cycli ;

  index_offset = 257 ;

#  partition = 3^5 4^2 ;

#  count_to = 524288 ;


##   other constraints --- examples
##    (statements may appear in any relative order)

  true = vsum[ v_weight, 2, 4 ];

  false = psum[ p_weight, 0, 1 ];

  true = elink[ -1, -3, incl, 1, 1 ];

  false = plink[ -2 ];
```

Part III — The model-file

23. A basic language for describing models

NB: This section conforms to the latest specifications, for use with `qgraf-4.0` — an appropriate version of this guide should be used for any previous version.

The input model should be described in the *model-file* (to be supplied by the user). A model-file is divided into several implicit sections (the *zones*), of which only two are always required. This section discusses only part of the language that can be used to describe input models; the additional statements are discussed in the next section. The first required zone contains the propagator declarations, and the second one the vertex declarations. For example, a model-file for Quantum Electrodynamics might look like this

```
%␣␣propagators
␣␣[electron,␣positron,␣-1]
␣␣[photon,␣photon,␣+1]
␣
%␣␣electromagnetic␣vertex
␣␣[␣positron,␣electron,␣photon␣]
```

The symbol ␣ is a *visible space*, which is used here to render the lines of the file clearer. There is no '`field` statement' — the fields are declared implicitly in the propagator statements. In this example there are three fields (denoted by the identifiers `electron`, `positron`, and `photon`), two propagators, and one (cubic) vertex.

Distinct statements should occupy distinct lines. A statement can occupy several lines provided the following conditions are met:

- neither empty lines nor annotations exist between the first and the last line occupied by the statement;
- the line breaks are consistent — that is, every (non-encoded) 'special string' (keyword, identifier, and so on) is contained in a single line;
- the character '\' (backslash) ends any line (except the last) containing only part of the statement (Section 3.4).

23.1 Propagators

The basic propagator statement is of the form

```
[ phi_1 , phi_2 , q ]
```

where `phi_1` and `phi_2` are fields and `q` is the commutation number ('`+1`' for fields satisfying commutation relations, '`-1`' for anti-commutation relations); the plus sign should not be omitted. Nevertheless, it is still possible to use just '`+`' and '`-`' if done consistently in each model-file (that is, only one convention can be used). The field `phi_1` is the conjugate of `phi_2` and conversely. If `phi_1` and `phi_2` are identical identifiers then we have a *neutral* or *self-conjugate* field, else `phi_1` and `phi_2` are dubbed *charged* fields. Sometimes `phi_1` and `phi_2` are not really particle and anti-particle, as in the case of ghost fields; nevertheless `phi_1` and `phi_2` will be called (respectively) the *particle* and the *anti-particle*, in an absolute sense.

The propagator represents a non-trivial contraction — ie vacuum expectation value of the time ordered product of two fields. Graphically, it is simply a type of (possibly oriented, (bi-)coloured) edge that can be used to construct diagrams. The above propagator declaration represents the contraction $<\phi_1 \phi_2>$, not $<\phi_2 \phi_1>$, and this is specially relevant in the case of anti-commuting fields. In the above example, and denoting the `electron` field by ψ, the propagator `[ electron, positron, -1 ]` corresponds to the usual $<\psi \bar{\psi}>$ contraction.

23.2 Vertices

The basic vertex statement (for declaring an interaction vertex of degree n) is of the following form.

```
[ phi_1, phi_2, ... , phi_n ]
```

Obviously, every vertex should have an even number of anti-commuting fields. Interactions are usually cubic and/or quartic, but in some cases (eg exotic gauges, effective models) there will be interactions of higher degree. Qgraf will accommodate for most of that: degrees in the range 3–8 are accepted by default.

The field ordering in the vertex declarations should not be neglected either. The above (generic) vertex declaration should represent a term (or set of terms) of the Lagrangian density in which the fields appear in that same order $\phi_1 \phi_2 \ldots \phi_n$, and the Feynman rule to be used when processing the diagrams should also match that ordering. In the case of the previous model-file, the vertex `[ positron, electron, photon ]` corresponds to the usual $\bar{\psi}\psi A_\mu$ ordering.

NB: Any mismatch between propagator and vertex declarations (on one side) and Feynman rules (on the other) is potentially disastrous — see in particular Section 38.

23.3 Functions (parameters) and constants

The syntax presented up to this point is the basic, minimal syntax. It lets us set the combinatorial description of the model (ie what kind of edges exist and how they are allowed to meet at the vertices of the diagrams), and whether the fields satisfy commutation or anti-commutation relations. That syntax must be extended to provide the ability to define parameters like spin, electric charge, mass, or even more complex objects representing eg Feynman rules.

Qgraf allows users to define functions for fields, propagators and vertices. Each such function — which maps either fields, propagators, or vertices to character strings — is represented by its own (unique) identifier. A function f that has a finite domain can be defined by a list of assignments of the form $x_k \to f(x_k)$, without using a generic 'formula' (not unlike a tabular definition), and that is essentially the method that is used. Moreover, it is also possible to define *constants*, which have a fixed 'value'. Both functions and constants should be declared before defining propagators and vertices. The constants should be declared in a single statement, eg

```
[ constants :: c1, c2, model ]
```

The functions come in three types, and there is a distinct statement for declaring each type.

For example, the statements

```
[ f_function :: ff1 ]
[ p_function :: pf1 ]
[ v_function :: vf1 ]
```

declare `ff1`, `pf1`, and `vf1` as (respectively) a field-function, a propagator-function, and a vertex-function. These are generic functions, which are treated as character strings. The `integer` qualifier should be added[11] when declaring functions that may be involved in some constraint (implemented in terms of the control-file statements `vsum` and `psum`), eg

```
[ integer v_functions :: vf2, vf3 ]
```

The functions of the same type should be declared using at most two statements — that is, *one* statement for the generic functions and *one* other statement for the integer functions, with no repeated identifiers between them.

A suitable modification of the model-file presented earlier will be used to illustrate these features. Here is the new version, this time without visible spaces.

```
%  constants
  [ constants :: model ]
  [ model = 'qed' ]

%  functions
  [ f_function :: C ]
  [ p_function :: m ]
  [ integer v_function :: g_power ]

%  propagators
  [ electron, positron, -1 ; C= ('-1', '+1'), m= 'me', s= '1/2' ]
  [ photon, photon, +1 ; C = ('0'), m= 'm0', s= '1' ]

%  electromagnetic vertex
  [ positron, electron, photon; g_power= '1' ]
```

In this example there is a single constant (ie `model`, whose value is '`qed`'), and there are three functions, one of each possible type. The f-function `C` maps `electron` to '`-1`', `positron` to '`+1`', and `photon` to '`0`' (one may think of `C` as the electric charge); the p-function `m` (which might stand for *mass*) maps the fermionic propagator to '`me`', and the bosonic propagator to '`m0`', whilst the p-function `s` (the spin, let us say) maps those propagators to '`1/2`' and '`1`', respectively. Finally, the v-function `g_power` maps the single vertex to '`1`' (this function may represent the power of the coupling constant appearing in the Feynman rule).

A v-function can be defined by including, in the declaration of each vertex, the image of that vertex under that function (see below for the exact notation), and a similar statement applies to p-functions. Defining an f-function is just slightly less trivial: every propagator

[11] This is not being enforced straight away, but it should be enforced in future versions.

declaration should contain either a single image or a pair of images, according to the number of distinct fields in that propagator. The images should be defined on the right-hand side of the propagator and/or vertex declarations, which should be separated from the left-hand side by a semicolon. As it should be clear by now, the syntax for defining the images of vertices and/or propagators is

```
function_id = S
```

where `function_id` is the function identifier and `S` is the image — an encoded string or, if allowed, an unencoded string (see below). In the case of f-functions the syntax is either

```
function_id = ( S1 , S2 )
```

if the propagator fields are charged, or

```
function_id = ( S )
```

in the opposite case (`S`, `S1`, and `S2` denote (un)encoded strings). Clearly, a p-function is a special case of an f-function, and can always be rewritten as such.

The following simplification is allowed in the definition of functions and constants: an image can be written unencoded whenever it is a valid identifier, integer, or rational (length allowing). Thus, some of the statements of the previous model description can be simplified as follows.

```
[ model = qed ]
[ electron, positron, -1 ; C= (-1, +1), m= me, s= 1/2 ]
[ photon, photon, +1 ; C = (0), m= m0, s= 1 ]
[ positron, electron, photon; g_power= 1 ]
```

Here are also some examples of definitions where that kind of simplification is not permitted: `x = ''`, `s = ' 1'`, `sum= '2+1'`, `key= 'a b'`, `v = '(0)'`, `rp = ')'`.

23.4 Optional keywords

There are a few additional keywords that serve to further characterize the fields of the model. The keyword `notadpole` prevents the generation of diagrams containing one-point insertions of the field(s) declared in the statement where that keyword is used. For instance, if a propagator is defined as

```
[ photon, photon, +1, notadpole ; \
  C = (0), s = 1 ]
```

then diagrams with tadpoles of the field `photon` will be systematically suppressed (this means that no such propagator will have an identically null momentum).

Sometimes one might want to declare fields that act simply as external sources; others, to suppress propagators with very large masses. In such cases, the keyword `external` can be used. The statement

```
[ Phi, Phi, +1, external ]
```

leads to the exclusion of the diagrams with propagators of the field `Phi`. Objectively, such statements do not declare propagators, only fields.

In contrast, the keyword `internal` can be used to specify that some fields should never appear as external fields of the scattering process (or correlation function) given as input. For example, if the model-file contains the propagator statement

```
        [ phi1, phi2, +1, internal ]
```

then `phi1` and `phi2` should both be excluded from the `in` and `out` statements, otherwise an error condition will occur.

The keyword `external` is incompatible with the other two.

24. An extended language

The input-model description has been overhauled recently, and it is now possible to define *sectors*, *submodels*, and *default values* (or simply *defaults*) for functions. What can be achieved with this extended language includes (*i*) a more compact description of 'complex' models (that is, models with many propagators and/or vertices, and for which one or more functions have been defined), and (*ii*) a combined description (in a single file) of a given model and some of its submodels and/or extensions. Creating a combined model-file can be a bit more time-consuming than creating separate files but, should some changes be required later on (eg adding new functions, or fixing some definition), it would then be possible to update all of those models by editing just one file.

A 'modern' model-file is divided into six *zones* (or implicit sections), though some of them may be empty. An illustrative example of such a file — that may be useful in inferring the sequence in which the different types of statements should appear — can be found in Section 27.

24.1 Sectors

A *sector* is a 'piece' of the model-file delimited by two statements of the form

```
sector[ sector1 ]
       ⋮
end[ sector1 ]
```

Here, `sector1` is the name of the sector, which must be an identifier. Not every type of statement may appear in a sector block — in fact, there are only two types of sectors, each of which includes at most two (other) types of statements. A *propagator-type* sector includes one or more propagator statements, possibly preceded by other statements (to be discussed later) that define default values for some f-functions and/or p-functions, eg

```
sector[ sector1 ]
  [ ; pf1= -1 ]
  [ phi, phi, +1 ; pf2= 'A' ]
  [ psi, psi, -1 ; pf2= 'B' ]
end[ sector1 ]
```

Likewise, a *vertex-type* sector should include one or more vertex statements, and may include additional statements that define default values for some v-functions, eg

```
sector[ sector4 ]
  [ ; vf1= 0 ]
  [ phi, phi, phi ]
  [ psi, psi, phi ]
end[ sector4 ]
```

Although propagator and vertex statements need not be part of a sector block, propagator statements should still precede vertex statements and (thus) propagator-type sectors should precede vertex-type sectors. The sectors are declared in zone 1, and there should be a single statement for each type of sector, eg

```
[ p_sectors :: sector1, sector2 ]
[ v_sectors :: sector3, sector4, sector5 ]
```

24.2 Submodels

The existence of sectors makes it possible to define *submodels* by instructing the program to treat some sectors as an integral part of the input model and other sectors as alien. The propagator and vertex statements that are not part of any sector are always part of the definition of every submodel. The submodels are declared at the very top of the model-file (zone 1), and then defined one by one in zone 4. For example, the statement

```
[ submodels :: subm1, subm2 ]
```

declares two submodels, and the submodel block

```
submodel[ subm1 ]
  [ include :: sector1, sector3 ]
end[ subm1 ]
```

defines submodel `subm1` by listing the sectors that it comprises (their order of appearance in the `include` statement being immaterial). A submodel can also be defined by listing the sectors that are not part of that submodel, eg

```
submodel[ subm2 ]
  [ exclude :: sector5 ]
end[ subm2 ]
```

For example, a submodel that comprises every sector of a model-file can be defined by either listing all the sectors in an `include` statement or using a *'void'* `exclude` statement, ie

```
[ exclude :: ]
```

Such a submodel *has* to be defined if (i) the complete model-file describes a (sub)model that the program should be able to use and (ii) at least one other submodel is defined.

Concerning the control-file, one may have either the usual kind of `model` statement if no submodel is defined in the model-file, eg

```
model = 'qed' ;
```

or else an extended statement that also specifies the submodel to be selected (in the next statement, submodel `qed1`), eg

```
model = qed1 // 'qedx' ;
```

24.3 Constants

Constants were introduced earlier. Still, when the model-file defines two or more submodels, it may be useful to define *submodel-dependent* constants. For example, one may wish to define for each submodel a distinct string representing the name of that submodel. The submodel-independent (or *global*) constants should be defined in zone 2, after (ie below) the respective declaration statement(s). This is illustrated in the next example (continued further below), where c1 is global and c2 is not (assuming it is not defined in zone 2).

```
[ constants :: c1, c2 ]
[ c1= 'g' ]
```

The submodel-dependent constants should be defined in every submodel block, unless defaults are defined (this alternative method is described in the next subsection). The syntax is as to be expected, eg

```
submodel[ subm1 ]
  [ include :: sector1, sector3 ]
  [ c2= 'qed3' ]
end[ subm1 ]

submodel[ subm2 ]
  [ exclude :: sector5 ]
  [ c2= 'qed1' ]
end[ subm2 ]
```

The first statement in a submodel block should be an `include` or an `exclude` statement. These are the only two types of model-file statements that can be *void*.

24.4 Default values

Suppose one wants to define a function that has many identical images — in other words, a highly non-injective function. In that case, it may be convenient to define defaults for that function. Default values can be defined globally and/or per sector, with the sectorial defaults overriding the global ones. Sectors need not be defined (just) for this purpose if all defaults are global.

For each function, it is possible to define at most one global default, and at most one default per sector. Defaults for distinct functions (and constants) should be defined in distinct statements. For example, the statement

```
[ ;; vf2 = 1 ]
```

(which should appear in zone 3, after declaring `vf2` as a vertex function, say) defines a global default for that function. As a result, `vf2` no longer has to be defined for every vertex — only those values that differ from that default have to be set explicitly, in the usual way. Additionally, if the model-file contains sectors then sectorial defaults can be defined too, whether or not global defaults exist, eg

```
sector[ sector3 ]
  [ ; vf2 = 0 ]
      ⋮
end[ sector3 ]
```

It then follows that `vf2` will be equal to `0` for every vertex in sector `sector3` unless that default is overridden by explicit definitions in the vertex statements. Having statements with either a single or a double semicolon may serve as a reminder that the corresponding defaults can be overridden at one or two other stages, respectively. Defaults for propagator-functions can be defined similarly, but for field-functions there are more possibilities. For example, the statement

```
[ ;; ff1= (0), (-1,1) ]
```

defines a global default for `ff1`, both for neutral and for charged fields. Nevertheless, it is possible to define partial defaults, eg

```
[ ;; ff1= (1) ]
```

or

```
[ ; ff1= (0,1) ]
```

since not every model (or sector) has both types of fields.

Within each sector block, the precedence rules are as follows: statements defining defaults for f-functions or p-functions should precede propagator statements, and statements defining v-function defaults should precede vertex statements. Thus, the last statement of any sector (excluding the `end` statement that closes the sector block) should be either a propagator statement or a vertex statement.

Furthermore, it is possible to define defaults for submodel-dependent constants, even though that feature is probably not very useful. In any case, if the statements

```
[ constants :: c2 ]
[ ; c2= 'qcd' ]
```

are included in zone 2 then '`qcd`' becomes the global default for `c2`. That default can be overridden in any submodel block, eg

```
submodel[ subm1 ]
  [ include :: sector1, sector3 ]
  [ c2= 'qcd1' ]
end[ subm1 ]
```

24.5 Additional remarks

Defining submodels requires introducing sectors as well, but not conversely. Note also that sectors and submodels should use distinct identifiers. Let us dub a sector as *active* when it is part of the submodel specified in the control-file, and as *alien* otherwise; when no submodel is defined then any sector is active. The active sectors are those that the program will focus on, of course; they are fully checked for syntax and content while the alien sectors are only partly checked. To properly debug a model-file it is necessary to run the program for every submodel defined in that file.

Although it is easy to create rather cryptic model-files with this extended language, that is likely not a very good idea; for instance, model-files may have to be edited (long) after coming into existence, possibly by someone who did not create the original file. The proactive measures to avoid creating such files include adding an adequate amount of commentary, avoiding cryptic identifiers, defining the sectors carefully, and grouping together those propagator (and vertex) statements that are not part of any sector (or even creating all the necessary sectors to preclude the existence of such statements).

NB: Should you decide to update your 'old' model-files, please note that they may still be used as input not only for some previous version of the program but also for QGRAF-R (until it is updated, at least, which should probably happen in 2024).

25. Duplicate vertices

A model has *duplicate vertices* if its description contains at least two vertex statements with the exact same fields and multiplicities. Each vertex may in fact have several copies, and although the corresponding statements may be identical that needs not be the case — the field ordering and the values of the v-functions may differ.

Duplicate vertices had been tolerated for some time but typically QGRAF generated more diagrams than necessary (with appropriate 'symmetry factors', though, so that the corresponding calculation could have been carried out). This has been addressed with the release of `qgraf-3.3` — additional symmetries are taken into account, so that there are usually fewer diagrams now.

26. The propagator commutation number

The usual propagator statement involves a *commutation sign*, eg

```
        [ phi, phi, + ]
        [ psi, psi, - ]
```

which may be referred to as either the propagator-sign or the field-sign. Nonetheless, to make this type of description look a bit more ‘algebraic’, it is now possible (starting with `qgraf-3.6`) to replace the commutation sign by a *signed commutation number*, eg

```
        [ phi, phi, +1 ]
        [ psi, psi, -1 ]
```

The plus sign should not be omitted. Also, there must be some consistency — in any given model-file, either commutation signs or commutation numbers should be used (but not both).

27. An example of a modern model-file

```
%     zone 1
%
%  the submodels first, then the sectors

  [ submodels :: subm1, subm2 ]

  [ p_sectors :: sk1 ]
  [ v_sectors :: sk2, sk3 ]

%     zone 2
%
%  the constants, the definitions of the global constants,
%   and the defaults of the non-global constants

  [ constants :: c1, c2 ]
  [ c1 = 'f' ]

%     zone 3
%
%  the functions and their global defaults

  [ integer f_functions :: ff1 ]
  [ p_functions :: pf1, pf2 ]
  [ ;; ff1= (-1,1), (0) ]
  [ ;; pf1= 1/2 ]
  [ integer v_function :: vf1 ]

%     zone 4
%
%  the submodel blocks, including the
%   definitions of the non-global constants

  submodel[subm1]
    [ include :: sk2 ]
    [ c2 = 'scalar submodel' ]
  end[subm1]
```

(continues on the next page)

```
  submodel[subm2]

    [ exclude :: ]

    [ c2 = 'full model' ]

  end[subm2]


%     zone 5   (required)
%
%  the propagators and the propagator-type sectors

  [ phi, phi, +1 ; pf1= 0, pf2= 'A' ]

  sector[sk1]

    [ ; pf2= 'B' ]

    [ psi, psibar, -1 ]

    [ lambda, lambda, -1 ; ff1= (1) ]

  end[sk1]


%     zone 6   (required)
%
%  the vertices and the vertex-type sectors

  [ phi, phi, phi ; vf1= 0 ]

  sector[sk2]

    [ phi, phi, phi, phi ; vf1= 1 ]

  end[sk2]

  sector[sk3]

    [ ; vf1= -1 ]

    [ psibar, psi, phi ]

    [ psibar, psi, lambda, lambda ]

  end[sk3]
```

Part IV — The style-file

28. Intrinsic representation of diagrams

This section presents a number of technicalities that will be needed for understanding and controlling the output of the program. To begin with, let us present some terminology. Apart from Feynman diagrams — viewed as pure combinatorial objects — sometimes we will also consider the underlying graphs, as if the Feynman diagrams had been deprived of their fields. When referring to those graphs we will use the terms *node* and *edge*. The external nodes are the nodes of degree one associated with the external fields, while the remaining nodes are called internal nodes. Similarly, an external edge is an edge that is incident to an external node, and any other edge is called internal. When referring to a Feynman diagram we will use the terms *vertex* and *propagator*; these terms are the analogue of internal node and internal edge, respectively, but they are meant to include the information about the attached fields.

The representations of a Feynman diagram that can be obtained in the output-file are based on a set of indices that label the basic components of the diagram. When generating a diagram QGRAF assigns one or more labels (integer numbers, let us stress) to each of the following objects: vertices, propagators, external fields, and internal fields. Terms like *vertex-index* and *propagator-index* will be used to denote the various labels associated with the diagram components.

A critical issue should be clarified at once: the objects that are labelled are (strictly) not the ones defined in the model-file. In that file, one may find a certain number of fields, propagators, and vertices that are considered to be different either because the strings that define them are different (in the case of fields) or because they involve a different set of fields (in the remaining cases). The labels we have just mentioned are given to *embedded* objects, ie attached to some graph component.

Let us see in more detail how one can define the embedding of fields, propagators, and vertices. Most of those cases are easy: propagators are attached to internal edges, vertices to internal nodes, and external fields to external nodes. What about the internal fields? Let us recall that in the perturbative expansion the interaction vertices supply the fields that are to be contracted in pairs, and which form propagators. Hence internal fields should be attached to objects surrounding the internal nodes. We could, for example, insert two auxiliary nodes into every internal edge and then attach the internal fields to those auxiliary nodes (as illustrated in Fig. 8a). There is no need for auxiliary nodes in the external edges — the external nodes will do.

The above mentioned method of field embedding is not unique. One could attach them to edges instead of nodes. External fields would be attached to external edges. One could insert an auxiliary node into every internal edge (therefore splitting every such edge into two) and then attach the internal fields to the resulting ‘half-edges’. Hereafter we will take for granted that the field embedding is properly defined, without relying too much on the actual method.

28.1 The basic indices (or labellings)

If a diagram has V internal nodes and P internal edges then QGRAF numbers its vertices from 1 to V, and its propagators from 1 to P — see the examples given in Fig. 7, diagrams

(a) and (b). Those labellings define what we will dub the vertex-index and the propagator-index, respectively.

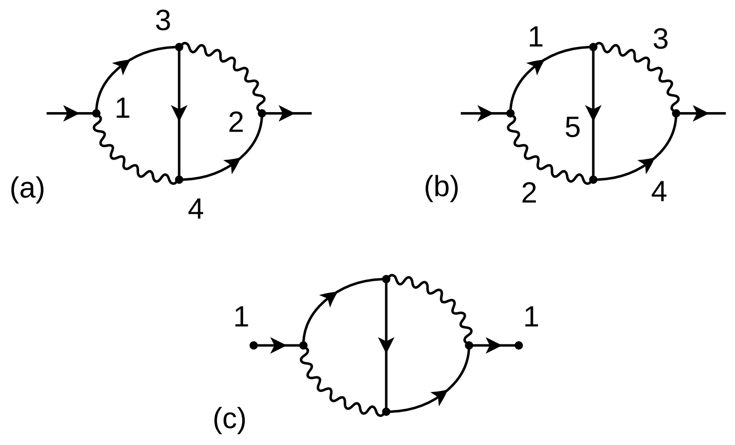

Fig. 7. Some indices for a simple diagram: (a) the vertex indices, (b) the propagator indices, and (c) the leg indices.

The embedded external fields — or legs — should be labelled too. There are two different leg indices, one for incoming fields (the *in-index*) and another for outgoing fields (the *out-index*). If a diagram has r incoming legs and s outgoing legs then the former receive the labels 1, 2, ... r and the latter the labels 1, 2, ... s. The label that is chosen for each leg follows automatically from the order in which the external fields were declared in the control-file. For example, if the external fields are declared by means of the following statements

```
in = positron[p1], electron[p2] ;
out = higgs[q1], muon_minus[q2], muon_plus[q3] ;
```

then the leg `positron` is assigned the in-index 1, and the leg `electron` the in-index 2; also, the leg `higgs` is assigned the out-index 1, the leg `muon_minus` the out-index 2, and the leg `muon_plus` the out-index 3.

QGRAF uses six basic labellings (indices). The field and the ray indices — the last two types of indices to be presented — are defined over the set of embedded fields.

The field-index uses the propagator and the leg indices. If a propagator has propagator-index k then its two fields receive the field-indices $2k-1$ and $2k$ (see Fig. 8b); if the particle differs from the anti-particle then the former gets the index $2k-1$ and the latter the index $2k$. An external field is assigned a negative index related to the leg index of the corresponding external node. Specifically, the field-index of an incoming (respectively, outgoing) field that has in-index (respectively, out-index) equal to j is defined as $-2j+1$ (respectively, $-2j$). This means that the incoming fields receive odd indices (-1, -3, ...) and the outgoing fields receive even indices (-2, -4, -6, ...). Although this labelling may seem unnatural at first, it allows one to distinguish external fields from internal ones — as well as incoming from outgoing fields — without reference to any other quantity.

The sixth and last type of labelling will be called *ray-index* because we may associate (visually) a propagator emerging from a vertex with a ray. For every vertex, the ray-index labels the surrounding vertex fields with the numbers 1, 2 ... , D (D being the degree of the interaction), an example of which is given in Fig. 8c. In contrast to other labellings, here labels differ only within each vertex; globally, there usually are repeated labels. The ray-index is not totally arbitrary: the index of an embedded field always coincides with the position (or one of the positions) of the field name in the definition of the respective vertex given in the model-file. For instance, if there is a vertex of the form

```
[positron, electron, photon]
```

then, for vertices of this type, the field `positron` will always be assigned the ray-index 1, the field `electron` the ray-index 2, and the field `photon` the ray-index 3, which means that in this case the labelling is unique — see Fig. 8(c). If the vertex includes repeated fields, some arbitrariness remains.

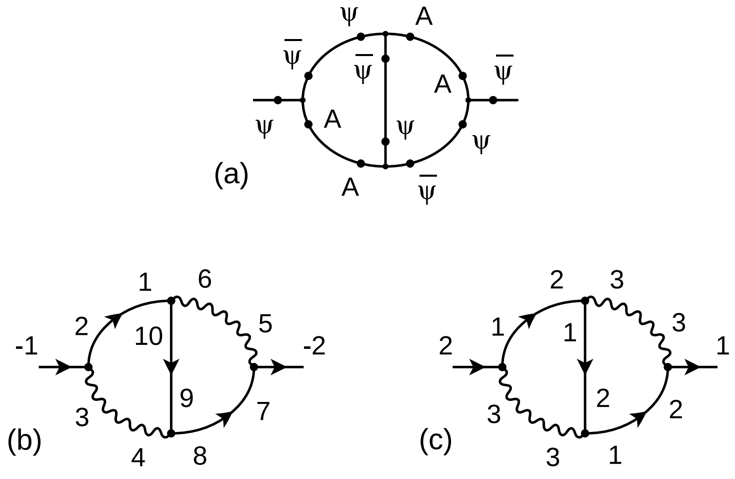

Fig. 8. Revisiting the diagram presented in Fig. 7: (a) a way of embedding the internal fields using auxiliary nodes, (b) the field indices, and (c) the ray indices.

We now have at our disposal two different notations for the embedded fields of a diagram. The first of these is a single-index notation: Φ_i denotes the field with *field-index* i. There is also a two-index notation: $\Phi_{i,j}$ denotes the field that belongs to vertex i (ie the vertex whose *vertex-index* is equal to i) and whose *ray-index* equals j. In the case of Fig. 8 one has $\Phi_{-1} = \Phi_{1,2} = \psi$ and $\Phi_6 = \Phi_{3,3} = A$.

Some of the indices presented in this section — like the vertex-index — are not completely determined in terms of a specific and complete rule. Hence one cannot predict eg the vertex-index of the vertices of most diagrams, relying on this guide only. Should the undocumented rules used internally by the program change in the future, no problem should arise provided no special property is assumed other than the generic ones herein presented.

28.2 The field-sign and the field-type

The *field-sign* is defined as '`+`' for commuting fields and '`-`' for anti-commuting fields. This quantity may also be referred to as the *propagator-sign*.

Necessarily for embedded fields, we will also define the *field-type*. It takes only three values, namely 1 (for incoming fields), 2 (for outgoing fields), and 3 (for internal fields).

28.3 The propagator orientation

The program also provides a set of symbolic expressions for representing the momentum flow, a feature that may be rather handy. It is obvious that in order to specify the momenta throughout the diagram we will have to choose a reference direction for every propagator. What will be called the *propagator momentum* is the momentum flowing in that direction.

Let us assume that the field embedding is properly defined, and that the field-index of every embedded field is known. The actual rule for defining the propagator orientation is as follows: we pick the direction in which, travelling along the propagator, the embedded field with field-index $2i$ is reached before the one with field-index $2i-1$. This coincides with the particle flow whenever the particle and the anti-particle differ.

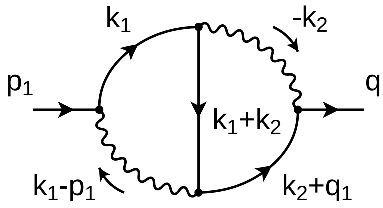

Fig. 9. The symbolic momenta for the diagram presented in Fig. 7.

In Fig. 9 one may see the symbolic momenta for the diagram presented in Fig. 7 (and in Fig. 8). It may be verified at once, by taking a look at Fig. 8(b), that the orientation of the propagators (indicated with arrows) is in agreement with the above mentioned rule for the field-indices.

29. Generating an output-file — introduction

In the initial versions — before the release of `qgraf-2.0` — there was just a short list of predefined output formats which could be used to generate the output-file. Having a fixed number of predetermined formats is a self-limiting approach. Since every program that may be used to process the output of a diagram generator is most likely bound to have its own notation, the number of formats, and thus the number of subroutines, may have to increase over time. A possible alternative is that users write their own conversion subroutines, one for every program they think of using. Another problem with the fixed format approach is that it lessens the potential ability of the program to incorporate into the output the parameters (represented by functions and constants) that are part of the definition of the input model. If one wishes to have (i) a model-file where the parameters are chosen by the user and (ii) an output containing a selection of those parameters, using a notation also chosen by the user, then the fixed format approach is inadequate.

In later versions there is a lot more flexibility, since the output of the program can be shaped with the help of a simple programming language. It is not just a matter of choosing the type of delimiters, spacing and similar marks — one can also choose what kind of information should appear in the output (there are some limitations, of course). In practical terms, it goes like this: to get a new format a user has to provide a file (the style-file) containing a rather short *program*, and that is all there is to it. Users can have a small collection of such files, and use different formats on different occasions. Typically, it seems much easier to write and (specially) modify such a file than a full conversion routine. It may be impossible to write a style-file that formats the output exactly like one wants; however, it should be possible to write a style-file in such a way that the output-file can be processed directly by one's favourite computer algebra system.

29.1 The sections of the style-file

In simple terms, the output-file can be divided into three parts. It starts with a *prologue* that may contain, for example,

- the name and version of the program that generated that file, and the approximate time the file was created;
- information that can be used to identify the input scattering process and the type of diagrams described in the file (eg the statements included in the control-file);
- extra information to be used by another program that will read the file (eg special marks to signal the beginning of the diagram list).

After that the diagrams are listed one by one, following a specific pattern. Finally there is a third section, dubbed the *epilogue*, whose basic purpose is to mark the end of the list and/or the end of the file; for example, it is important to know that the program did not exit prematurely. There is also a second opportunity to pass on additional information to the program(s) that will process the output-file. Each one of those three output sections is defined in a corresponding section of the style-file.

The syntax of the style-file is completely different from any syntax discussed so far. Apart from annotations — see below — the style-file contains 'text' (ie printable `ASCII` characters that, with a few exceptions, are taken literally) as well as 'keywords' (*style-keywords*).

An intrinsic style-keyword is a reserved string that starts with '`<`' and ends with '`>`', eg `<end>`. The keywords that delimit the specification of the output sections, and which must appear in any style-file, are the following.

```
<prologue>
<diagram>
<epilogue>
<exit>
```

They should always appear in that specific order, and only once, each one on its own line (containing no further characters) and starting at the leftmost position. Those four keywords are the only ones that must appear in any style-file (note that a string like `<exit>` is not a keyword if it is part of some annotation). Apart from syntactic requirements every other keyword is optional — although a style-file containing no keyword other than those required keywords is not terribly useful.

The prologue specification starts on the line following the `<prologue>` keyword and ends on the line that precedes the `<diagram>` keyword. Analogously, the diagram section specification is bounded by the lines containing the keywords `<diagram>` and `<epilogue>`, and the epilogue specification comes between the lines defined by the keywords `<epilogue>` and `<exit>`. What comes either before the line with the keyword `<prologue>` or after the line containing the keyword `<exit>`, if anything, must consist of either empty lines or annotations — but the latter have to conform to some rules (see Section 3.3). Elsewhere (ie in any of the three sections of the style-file) no annotations are allowed.

QGRAF reads the style-file(s) before it starts generating diagrams, and stores internally what it read. It can then build the character strings to be written to the output-file (or passed on to the program that calls the API). Each such string, initially empty, has a fixed end (the 'left-end', so to say, which is a kind of starting point) and another end (the 'right-end') that can change during the computation. An output string is built by following a sequence of two basic operations: appending some characters to the non-fixed end of the current string and, less frequently, deleting one or more characters from that same end. As the possibility of deleting characters exists, no such string is output before the section that defines it has been fully executed.

29.2 The prologue section

For the sake of illustration consider a fictitious style-file whose prologue section consists of the following lines.

```
#
#␣file␣generated␣by␣<program>
#␣␣<back>
#␣on␣<full_time>
#
<statement_loop><statement_sub_loop># <sub_statement>
<end><end>#
#
␣tsum␣:=␣0
```

This description is processed as follows. The program starts with an empty output string, as already mentioned, and then line 1 tells it to add a '`#`' as well as a *newline*,[12] which is always added when the end of a line is reached. Line 2 instructs the program to add another 20 characters (shown), then to perform the action determined by the keyword `<program>`, and finally to add another newline. That keyword produces a string that describes the name and the version of the program; what that string is can be deduced from the actual output (shown below) generated from the above example (and from a control-file whose contents may be inferred from that same output).

```
#
#␣file␣generated␣by␣qgraf-3.4.2
#␣
#␣on␣2023/08/14␣01:15:14.387␣+0100
#
#␣␣output␣=␣'d_list'␣;
#␣␣style␣=␣'sum.sty'␣;
#␣␣model␣=␣'qed'␣;
#␣␣in␣=␣electron[p1]␣;
#␣␣out␣=␣electron[q1],␣photon[q2]␣;
#␣␣loops␣=␣3;
#␣␣loop_momentum␣=␣k;
#␣␣options␣=␣floop,␣onepi␣;
#
␣tsum␣:=␣0
```

The keyword `<back>` deletes the rightmost character from the output string that is being built, which may generate an error if the string is empty at that point. That keyword may appear in any of the sections of the style-file but there should be no interference between different sections — nor, in the case of the diagram section, interference between different diagrams. Trailing spaces are ignored, no matter what type of input file is being read — there is usually some difficulty not only for a user to see them but also for Fortran to read them. Still, it is possible to output trailing spaces, as exemplified by line 3 of the previous style-file: after adding one hash character and two spaces, the program will delete the rightmost space occupying the position and then add a newline, leaving a trailing space. The keyword `<back>` may also be used to concatenate consecutive lines from the style-file. For example, the two input lines

```
This␣is␣a
<back>␣single␣line␣!
```

will be concatenated (the keyword excepted), since `<back>` will delete the newline that divides them (as long as the character '`<`' is in column 1). Thus, a long line can be split into two or more lines without introducing unwanted newlines in the output-file; one reason for doing this is that there is a maximum allowed length for any input line.

Next, line 4 instructs the program to add another 5 characters and then (owing to the presence of the keyword `<full_time>`) write the current date and time. The keywords `<full_time>` and `<raw_time>` are rather recent additions, described in Section 29.3.

Line 6 from our example is more interesting. The keyword `<statement_loop>` tells the

[12] To keep the discussion as simple as possible, let us assume that there is a newline character whose purpose is to mark the end of a line and implicitly the beginning of the subsequent line, should there be one.

program to perform a loop[13] (ie a programming loop); whatever is in between that keyword and the matching keyword `<end>` is executed once for every statement found in the control-file, following the same statement sequence (empty lines and annotations are ignored). In our example, there is a second style-loop nested within the first and defined by the keywords `<statement_sub_loop>` and the corresponding `<end>`. This inner loop is executed once for each line occupied by the statement that the outer loop defines implicitly (depending on the iteration step), following the line sequence from the control-file. Moreover, the keyword `<sub_statement>` tells the program to add the character string that corresponds to the statement line defined implicitly by the inner loop, and for the statement defined implicitly by the outer loop — iteratively, ie one string each time the inner loop is executed. That string includes neither the newline nor (if present) the statement continuation mark, nor any trailing spaces.

To sum up, the statements from the control-file can be written to the output-file by executing two nested style-loops and including the keyword `<sub_statement>` in the inner loop; the formatting is somewhat free. In our example, each statement from the control-file takes up a single line. Nevertheless, if one rewrites that file so that at least one statement occupies more than one line then the output-file may include something like this:

```
#␣␣output
#␣␣␣=
#␣␣␣␣'d_list'␣;
```

That may or may not be what one wishes to obtain; it should not be difficult to adjust this kind of output. To present another example, the next two lines of code will format statements occupying more than one input line into a single output line:

```
<statement_loop>#␣<statement_sub_loop><sub_statement>␣<end>
<back><back><end>#
```

It is no longer possible to print the control-file statements using a single style-loop. Nevertheless, anything that could be done with the earlier single loop construct should be possible to do with the two loop construct as well.

There are four exceptional cases for representing printable characters in the style-file, and they are as follows (the ASCII characters that have to be encoded are on the left-hand side, and their respective encodings on the right-hand side).

`<`	→	`<<`
`>`	→	`>>`
`[`	→	`[[`
`]`	→	`]]`

This means that those four characters must be duplicated in the style-file if they are to appear in the output-file. With this convention it is always possible to distinguish 'text' from keywords. The first two exceptions are related to the use of the characters '`<`' and '`>`' in the delimitation of intrinsic keywords. The square brackets play a similar role in the case of non-intrinsic keywords, which are derived from the identifiers of user-defined constants and functions. In fact there is an additional special case, namely that of a trailing space character, which has already been discussed. This case is a bit different, since it is about always reading and writing a certain character rather than having to encode it.

[13] This type of construct, of which the present kind is not the only example, will be dubbed a *style-loop*.

29.3 The style-keywords `<full_time>` and `<raw_time>`

The output produced by the keyword `<full_time>` should be the 'full time' (ie *date*, *time*, and *time zone*, in ASCII), as provided by the operating system — which means that the correctness of this output depends on having an appropriately configured environment. That output should look like this:

```
2023/08/14 01:15:14.387 +0100
```

The date format uses the sequence year/month/day, as it is the Fortran standard; the slashes and the colon signs are added by QGRAF. Alternatively, the keyword `<raw_time>` produces a similar output without added characters, eg

```
20240915 151401.783 +0100
```

The obvious usefulness of these keywords consists in enabling one to register the creation time in the file itself — the time set by the operating system can be changed involuntarily if one is not careful enough (eg when copying the file).

A very good estimate of the *real time* (or *wall clock time*) taken by the execution of the program can be obtained by inserting any of those keywords in both of the above mentioned sections and then computing the elapsed time from the respective outputs. That elapsed time might not (fully) include the period of time taken by the operating system in copying the output-file to the computer disk, as that file is usually created in the RAM and often not saved at once.

29.4 The epilogue section

The epilogue section may include every keyword allowed in the prologue section, and no other. Therefore, all the information about the input statements and the version of the program may be included in the epilogue section instead, or even in both sections.

Recent additions to the scope of the prologue and epilogue sections are discussed in Section 32.5. The style-keywords allowed in these sections are listed in Section 33. For obvious reasons the output produced by `<diagram_counter>` and `<diagram_index>` depends usually on which section these keywords are used.

30. The diagram section

The style-keywords can be divided into two main classes. One class contains what one may call control keywords; they serve to delimit the output sections, to define the style-loops, etc, but do not generate information by themselves. In this class we may find keywords like `<diagram>`, `<statement_loop>`, and `<end>`. The other class includes the keywords that instruct the program to append information to the output string; these will be called data keywords. We have already seen a few data keywords, namely `<program>`, `<sub_statement>` and `<full_time>`, but many more exist.

Data keywords may themselves be divided into local and global keywords. Local keywords are those that must be used in one of the style-loops, while global keywords have no such restriction. Global keywords do not have to be constants, for example the keyword `<diagram_index>` produces different strings at different stages.

Let us now discuss the diagram section, which is obviously the most important. There are many keywords that can be used in that section, most of which are data keywords.

30.1 The global keywords

The global keywords are as follows.

- `<diagram_index>` — a positive integer specifying the order in which a diagram was generated (ie `1` for the first diagram, `2` for the second diagram, etc), unless an index-offset has been defined (in which case that offset is added to the natural diagram index).
- `<legs>` — the number of external fields of the diagram
- `<legs_in>` — the number of incoming fields
- `<legs_out>` — the number of outgoing fields
- `<local_symmetry_number>` — see Section 32.7
- `<loops>` — the number of loops of the diagram
- `<loops1>` — the first (and possibly the only) value in the `loops` statement
- `<loops2>` — the second value in the `loops` statement, if it exists, else the only value
- `<minus>` — similar to `<sign>` (see below) if the diagram sign is minus, otherwise it produces an empty string
- `<nonlocal_symmetry_number>` — see Section 32.7
- `<propagators>` — the number of internal edges of the diagram
- `<sign>` — the diagram sign (either a plus or a minus sign) that follows from the anti-commutation rules
- `<symmetry_factor>` — the diagram symmetry factor (either 1, if there are no symmetries, or a fraction like $1/2$, or in general $1/n$)
- `<symmetry_number>` — the diagram symmetry number (a positive integer equal to the reciprocal of the symmetry factor)
- `<vertices>` — the number of internal nodes of the diagram

There are five main types of style-loops in the diagram section, and every one of them is optional. The keywords

```
<in_loop>
<out_loop>
<propagator_loop>
<vertex_loop>
```

announce four of those loops, and the keyword `<end>` closes them. Those style-loops are executed as many times as there are (respectively) incoming particles, outgoing particles, propagators, and vertices in the diagram being listed. During the execution of a style-loop the program prepares itself to examine the relevant class of objects defined by the loop (ie legs, propagators, or vertices), as well as some other adjacent objects, and then prints information about them if requested to do so. The fifth style-loop type is defined by the keyword `<ray_loop>` and it should always appear nested inside the vertex-loop, like this:

`<vertex_loop>` ... `<ray_loop>` ... `<end>` ... `<end>`

The ray-loop is needed to tell the program to examine every line incident with the vertex that is iteratively defined by the vertex-loop. Those five style-loops form the basic tool to access the local information that defines a diagram, where *local information* means that it refers either to the component of the diagram being examined, or to some neighbouring component. For instance, if the object being examined is a vertex v_0 then some information regarding the vertices adjacent to v_0 is also available at that time, as is the information on any propagators and external lines incident with v_0. However, at that same time, no information about other (more remote) objects is available, with the obvious exception of the information provided by the global keywords.

	i_loop	o_loop	p_loop	r_loop	v_loop
<dual-field>	●	●	●	●	
<dual-field_index>			●	●	
<dual-momentum>	●	●	●	●	
<dual-ray_index>			●	●	
<dual-vertex_degree>			●	●	
<dual-vertex_index>			●	●	
<field>	●	●	●	●	
<field_index>	●	●	●	●	
<field_sign>	●	●	●	●	
<field_type>	●	●	●	●	
<in_index>	●				
<leg_index>	●	●			
<momentum>	●	●	●	●	
<out_index>		●			
<propagator_index>			●	●	
<ray_index>	●	●	●	●	
<vertex_degree>	●	●	●	●	●
<vertex_index>	●	●	●	●	●

What remains to be explained here is which local keywords may be used inside which style-loops, as well as what the keywords stand for. The former of those issues is addressed in the above table. A given local keyword may be used in a certain loop type if and only if the respective table entry is marked with a full circle. The loop types have been abbreviated — `i_loop` stands for in-loop, `r_loop` for ray-loop, and so on. It is clear that if a keyword may be used in the vertex-loop then it may also be used in the ray-loop, although the information it represents will remain constant while the ray-loop is executed.

The exact definition of a style-keyword depends on the type of loop where it is used. The keywords allowed in the basic loop types are described next.

30.2 The propagator-loop

The propagator-loop is executed as follows: when the keyword `<propagator_loop>` is found the program assigns the value 1 to the loop index i, which coincides with the propagator-index, and prepares itself to examine the propagator with index equal to 1. Then it outputs the information requested about that propagator (and possibly some other information, or some fixed characters), until it reaches the keyword `<end>`. At that point it increments the loop index to 2, and the rest is easily guessed. The loop terminates when every propagator has been examined.

To infer what kind of information is available during the execution of the propagator-loop let us take a look at Fig. 10, which has been obtained by grouping together the figures included in Section 28 but retaining only part of the original diagram, namely propagator 5 and its neighbourhood.

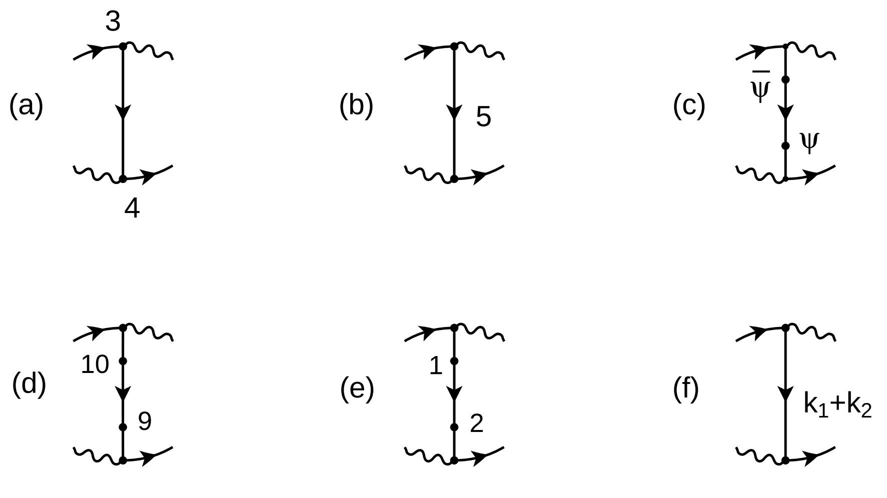

Fig. 10. Some local information available in the propagator-loop: (a) vertex indices, (b) propagator-index, (c) propagator fields, (d) field-indices, (e) ray indices, and (f) propagator momentum.

A list of the keywords allowed in the propagator-loop is given below. Each entry includes a keyword, the respective meaning, and an output string generated by that same keyword. That string is the one that would be obtained in the description of the propagator shown in Fig. 10 (the string is in parenthesis, after the arrow symbol), in which case we may identify ψ with `electron` and $\overline{\psi}$ with `positron`.

Recall that if a propagator is assigned a propagator-index i then the respective fields (to be noted Φ_{2i-1} and Φ_{2i}) will have field-indices equal to $2i{-}1$ and to $2i$. Generically, let $\hat{v}_j$ denote the vertex to which Φ_j belongs (hence j is a field-index, but not a vertex-index).

- `<dual-field>` — the name of Φ_{2i} ($\rightarrow$ `positron`)
- `<dual-field_index>` — the unsigned integer $2i$ ($\rightarrow$ `10`)
- `<dual-momentum>` — the opposite of `<momentum>` ($\rightarrow$ `-k1-k2`)
- `<dual-ray_index>` — the ray-index of Φ_{2i} (in its vertex) ($\rightarrow$ `1`)
- `<dual-vertex_degree>` — the degree of $\hat{v}_{2i}$ ($\rightarrow$ `3`)
- `<dual-vertex_index>` — the vertex-index of $\hat{v}_{2i}$ ($\rightarrow$ `3`)
- `<field>` — the name of Φ_{2i-1} ($\rightarrow$ `electron`)
- `<field_index>` — the unsigned integer $2i{-}1$ ($\rightarrow$ `9`)
- `<field_type>` — the field-type of Φ_{2i-1} ($\rightarrow$ `3`)
- `<field_sign>` — the sign of propagator i ($\rightarrow$ `-`)
- `<momentum>` — the momentum of propagator i ($\rightarrow$ `k1+k2`)
- `<propagator_index>` — the unsigned loop index i ($\rightarrow$ `5`)
- `<ray_index>` — the ray-index of Φ_{2i-1} (in its vertex) ($\rightarrow$ `2`)
- `<vertex_degree>` — the degree of $\hat{v}_{2i-1}$ ($\rightarrow$ `3`)
- `<vertex_index>` — the vertex-index of $\hat{v}_{2i-1}$ ($\rightarrow$ `4`)

30.3 The leg loops

There are two types of leg loops, the `<in_loop>` and the `<out_loop>`, which are executed once for each incoming (respectively, outgoing) field. The legs of the diagram are, rather obviously, the main objects that are accessed in those style-loops.

Let r and s be the number of incoming and of outgoing fields. Those fields appear in the control-file in a certain sequence, and the n^{th} incoming (or outgoing) field will be noted Φ^{in}_n (respectively, Φ^{out}_n).

To illustrate the keywords available for leg loops, we shall make use of a diagram that has been used before (in eg Section 28), whose legs are specified by the following statements.

```
        in = electron[pp1] ;
        out = electron[qq1] ;
```

Since $r{=}s{=}1$ (in our example), each keyword produces a single string (below). Consider first the `<in_loop>`, where the loop index i runs from 1 to r. Let v_k be the vertex that leg i is incident to, and j the ray-index (with respect to that vertex) of Φ^{in}_i. The field Φ^{in}_i can then be identified with $\Phi_{k,j}$.

- `<dual-field>` — the conjugate of `<field>` ($\rightarrow$ `positron`)
- `<dual-momentum>` — the opposite of `<momentum>` ($\rightarrow$ `-pp1`)
- `<field>` — the name of Φ^{in}_i ($\rightarrow$ `electron`)

- `<field_index>` — the field-index of Φ_i^{in} ($\rightarrow$ `-1`)
- `<field_type>` — the field-type of Φ_i^{in} ($\rightarrow$ `1`)
- `<field_sign>` — the sign of Φ_i^{in} ($\rightarrow$ `-`)
- `<in_index>` — the unsigned loop index i ($\rightarrow$ `1`)
- `<leg_index>` — the same as `<in_index>` (in this type of loop) ($\rightarrow$ `1`)
- `<momentum>` — the momentum flowing into the diagram through leg i ($\rightarrow$ `pp1`)
- `<ray_index>` — the ray-index of $\Phi_{k,j}$ (ie the unsigned integer j) ($\rightarrow$ `2`)
- `<vertex_degree>` — the degree of vertex v_k ($\rightarrow$ `3`)
- `<vertex_index>` — the index of vertex v_k (ie the unsigned integer k) ($\rightarrow$ `1`)

Some of the above definitions have to be changed for the `<out_loop>`. The loop index i goes from 1 to s, but the leg index goes from $r{+}1$ to $r{+}s$. The keyword `<field>` now produces outgoing fields, and the momentum direction points outwards. Thus $\Phi_{k,j}$ is now the *conjugate* of Φ_i^{out}.

- `<dual-field>` — the conjugate of `<field>` ($\rightarrow$ `positron`)
- `<dual-momentum>` — the opposite of `<momentum>` ($\rightarrow$ `-qq1`)
- `<field>` — the name of Φ_i^{out} ($\rightarrow$ `electron`)
- `<field_index>` — the field-index of Φ_i^{out} ($\rightarrow$ `-2`)
- `<field_type>` — the field-type of Φ_i^{out} ($\rightarrow$ `2`)
- `<field_sign>` — the sign of Φ_i^{out} ($\rightarrow$ `-`)
- `<leg_index>` — the unsigned integer equal to $r{+}i$ ($\rightarrow$ `2`)
- `<momentum>` — the momentum leaving the diagram through leg i ($\rightarrow$ `qq1`)
- `<out_index>` — the unsigned loop index i ($\rightarrow$ `1`)
- `<ray_index>` — the ray-index of $\Phi_{k,j}$ (ie the unsigned integer j) ($\rightarrow$ `1`)
- `<vertex_degree>` — the degree of vertex v_k ($\rightarrow$ `3`)
- `<vertex_index>` — the index of vertex v_k (ie the unsigned integer k) ($\rightarrow$ `2`)

30.4 The vertex-loop and the ray-loop

The vertex-loop tells the program to visit each vertex of the diagram, and the ray-loop to visit each edge (or possibly half-edge) incident with each such vertex. The indices associated to these loops are the vertex-index and the ray-index (in the following we will denote them by i and j, respectively).

A vertex v_i of degree d_i comprises the fields $\Phi_{i,j}$, for $j{=}1,\ldots d_i$. Fig. 11 illustrates the three possible cases: $\Phi_{i,j}$ can be an external field (left), an internal field that is part of a propagator joining two distinct vertices (centre), or an internal field that is part of a propagator built from two fields of the same vertex (right). If, for a given value of j, $\Phi_{i,j}$ is an internal field then there is a propagator P_m connecting v_i to another vertex v_k (or else connecting v_i to itself, in which case we just set $k = i$); that propagator also contains another field — belonging to v_k — which is the conjugate of $\Phi_{i,j}$ and that will be denoted by $\Phi_{k,l}$. Note that v_k and $\Phi_{k,l}$ are both undefined whenever $\Phi_{i,j}$ is an external field.

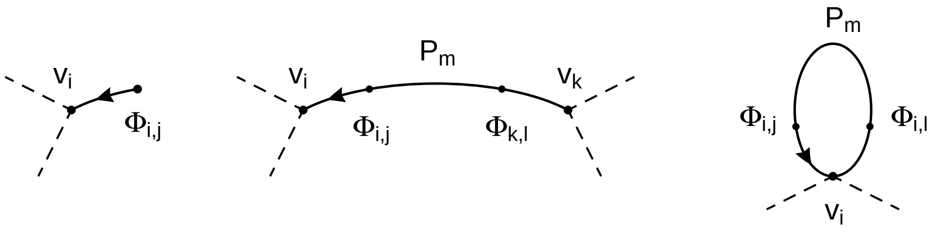

Fig. 11. Basic notation used in describing the vertex-loop and the ray-loop.

In the ray-loop the keyword `<momentum>` refers to the momentum flowing along the edge to which $\Phi_{i,j}$ is attached, in the direction that is shown graphically in Fig. 11 by means of arrows. This graphical rule gives the momentum flowing into vertex v_i coming from that edge, except that if $\Phi_{i,j}$ is an internal field and $i = k$ then this wording must be made more precise.

Fig. 12 includes part of the same diagram presented earlier — now it shows vertex 1 and its neighbourhood — and will also be used for illustrating the meaning of the keywords presented below. This time the output of the program given with each entry can be divided in two cases. For the keywords that do not require the use of the ray-loop the output string contains a single label, namely the one that corresponds to vertex 1. For the other keywords the output string is composed of three sub-strings (since the degree of vertex 1 is equal to 3), one for each ray-index; that means that the execution of the ray-loop is simulated, but the execution of the vertex-loop is not.

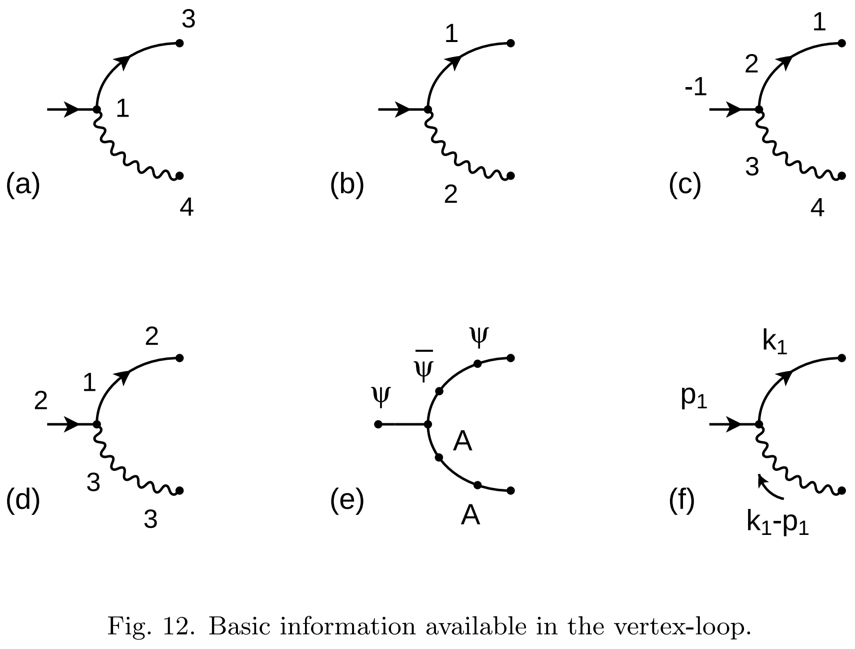

Fig. 12. Basic information available in the vertex-loop.

- `<dual-field>` — the name of the conjugate of field $\Phi_{i,j}$ ($\rightarrow$ `electron positron photon`)
- `<dual-field_index>` — the field-index of $\Phi_{k,l}$ (Fig. 11) if that field exists, otherwise zero ($\rightarrow$ `1 0 4`)
- `<dual-momentum>` — the opposite of `<momentum>` ($\rightarrow$ `k1 -p1 -k1+p1`)
- `<dual-ray_index>` — the ray-index of $\Phi_{k,l}$ (ie the unsigned integer l) if that field exists, else zero ($\rightarrow$ `2 0 3`)
- `<dual-vertex_degree>` — the degree of vertex v_k, if that vertex exists, else zero ($\rightarrow$ `3 0 3`)
- `<dual-vertex_index>` — the index of vertex v_k (ie the unsigned integer k), if it exists, otherwise zero ($\rightarrow$ `3 0 4`)
- `<field>` — the name of $\Phi_{i,j}$ ($\rightarrow$ `positron electron photon`)
- `<field_index>` — the field-index of $\Phi_{i,j}$ ($\rightarrow$ `2 -1 3`)
- `<field_type>` — the field-type of $\Phi_{i,j}$ ($\rightarrow$ `3 1 3`)
- `<field_sign>` — the sign of $\Phi_{i,j}$ ($\rightarrow$ `- - +`)
- `<momentum>` — the momentum flowing into vertex v_i coming from the edge with ray-index j ($\rightarrow$ `-k1 p1 k1-p1`)
- `<propagator_index>` — the propagator-index of P_m (Fig. 11), ie the unsigned integer m if it exists, else the field-index of the external field $\Phi_{i,j}$ ($\rightarrow$ `1 -1 2`)
- `<ray_index>` — the unsigned integer j ($\rightarrow$ `1 2 3`)
- `<vertex_degree>` — the degree of vertex v_i ($\rightarrow$ `3`)
- `<vertex_index>` — the unsigned integer i ($\rightarrow$ `1`)

Five keywords — `<dual-field_index>`, `<dual-ray_index>`, `<dual-vertex_index>`, `<dual-vertex_degree>`, and `<propagator_index>` — are defined in a peculiar way when the ray-index j defines an external line. This is done for two main reasons: it either ensures compatibility with earlier pre-defined formats or it provides more information than a straightforward definition would. The core issue is the same: either one does not accept any of those keywords as valid, or else one must define them *ad hoc* for the cases where a natural definition fails to exist (ie for vertices incident with external lines). An additional keyword, `<dual-field>`, is defined in a way that is not consistent with the meaning of the prefix `dual-` as used in other keywords; in addition, if it were consistent in that regard then it would suffer from the same problem that the other five keywords do. This problem will hopefully reach an acceptable status in a future release, as other features become available.

30.5 The style-keywords `<new_loops>`, `<new_partition>`, `<new_topology>` and `<new_elinks>`

These four style-keywords are not normal data keywords, as they usually depend not only on the current diagram but also on the preceding (output) diagram. They always produce a '`1`' for the first diagram of every run (this bit may not be repeated below). Note also the implicit hierarchy, which follows from the diagram generation algorithm.

The keyword `<new_loops>` may be useful when the `loops` statement requests diagrams for more than one *cycle-rank* (or *number of loops*). This keyword produces a '`1`' in the following cases (otherwise, it produces a '`0`'):

- when the current diagram is the first output diagram (whether or not diagrams for more than one cycle-rank have been requested);
- else, when the cycle-rank of the current diagram differs from the cycle-rank of the previous diagram.

The keyword `<new_partition>` produces a '`1`' in the following cases (otherwise, a '`0`' is produced):

- when `<new_loops>` produces (or would produce, if present) a '`1`';
- else, when the vertex degree partition of the current diagram differs from that of the preceding diagram.

The output generated by `<new_topology>` specifies whether or not the topology of the current diagram differs from that of the preceding diagram; here, *'topology'* means unlabelled topology, with no fields nor associated labels. Thus, a '`1`' is produced in the following cases (else a '`0`' is produced):

- when `<new_partition>` produces a '`1`';
- else, when the unlabelled topology of the current diagram differs from that of the previous diagram.

Finally, `<new_elinks>` refers to labelled topologies, ie to topologies with labelled external fields. This keyword produces a '`1`' in the following cases (else a '`0`' is produced):

- when `<new_topology>` produces a '`1`';
- else, when the configuration of the external fields in the current diagram differs from the corresponding configuration in the preceding diagram.

As an example of a potential application, one may point out that `<new_topology>` and `<new_elinks>` can be employed to speed-up the diagram topology identification. In addition, the information that the same topology is shared by multiple (specific) diagrams can often be exploited to improve the efficiency of the computations that involve large numbers of diagrams (sometimes even when no topology has been precisely determined).

On the other hand, if the model has a single (self-conjugate) field then `<new_elinks>` becomes irrelevant, as it always produces a '`1`'. Also, the usefulness of these keywords might be reduced to some extent, or even completely, if (say) the output diagrams were arbitrarily divided among several files, or (worse still) if those diagrams were sorted in a different way.

30.6 A first survey of output styles

The set of available keywords is certainly not a minimal one. Also, from a strict point of view, one does not need all of the existing types of loops. The advantage of this abundance is that one can choose the features that suit one best (for example, some features which are instantly available could require some extra programming if a minimal set were used).

The main issue concerning the choice of output style is the processing of the expressions built by QGRAF. The output styles defined by the files `sum.sty`, `array.sty`, and `form.sty` may serve to illustrate three different approaches. The first one consists in combining all the symbolic expressions into a single expression, that is, to add them up. Further processing may be problematic if the number of diagrams is large. In the second approach an array/vector is defined, each component storing the symbolic expression for a single diagram; then the processing program reads the whole array and manipulates all the expressions in a single run. In some cases this may still be problematic, or just inefficient. In the third approach, which has been used together with `FORM` [6], the expressions are still kept in a single file but each expression can be read and processed separately.

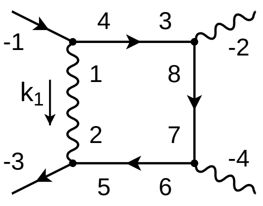

Fig. 13. A diagram from QED, and the corresponding field-indices (the embedded fields are implicit).

Let us now look at the kind of output produced from the style-files `array.sty` and `form.sty` that are part of (eg) the latest package. Although the corresponding output styles are outdated, a brief analysis of those styles can still be profitable if it provides a basis for building other output styles. The examples presented below refer to the diagram shown in Fig. 13; the file `array.sty` is used by the first one.

```
a(1):= (+1)*
pol(e(-1,p1))*
pol(p(-3,p2))*
pol(A(-2,q1))*
pol(A(-4,q2))*
prop(A(1,-k1))*
prop(e(3,-k1+p1))*
prop(e(5,-k1-p2))*
prop(e(7,-k1+p1-q1))*
vrtx(p(4,k1-p1),e(-1,p1),A(1,-k1))*
vrtx(p(-3,p2),e(5,-k1-p2),A(2,k1))*
vrtx(p(8,k1-p1+q1),e(3,-k1+p1),A(-2,-q1))*
vrtx(p(6,k1+p2),e(7,-k1+p1-q1),A(-4,-q2));
```

The numbers that appear in the above expression as arguments of the fields are the field-indices. Four types of style-loops are used. The propagators have a single argument since the second argument would not contain new information (the second field would be the conjugate of the first, the second index would be equal to the successor of the first, and the momentum would be the opposite of the first).

If the style-file `form.sty` were used instead, that same diagram would be described as follows.

```
*--#[ d1:
*
    1
   *vx(p(2),e(-1),A(1))
   *vx(p(-3),e(3),A(1))
   *vx(p(4),e(2),A(-2))
   *vx(p(3),e(4),A(-4))
*
*--#] d1:
```

Here only the propagator-index is used; the two fields with a common argument belong to the same propagator, and fields whose argument does not match are the external fields. This notation seems to be insufficient for models that include Majorana fields (the ones for which the particle and the anti-particle coincide and the field components anti-commute).

31. Non-intrinsic style-keywords

The previous section discusses intrinsic style-keywords and the kind of output it is possible to obtain from them. Nevertheless, since it is possible to define parameters for the fields, propagators and vertices (Section 23.3), one might want the diagram description to include such parameters. A non-intrinsic style-keyword is a keyword that derives from some (user-defined) constant or function. The subject of the present section is on how to define such keywords and how to include them in a style-file.

31.1 Keywords that derive from functions

Consider once more QED, described as follows.

```
%  propagators
  [ electron, positron, -1 ; prop= 'S', m= 'me']
  [ photon, photon, +1 ; prop = 'P', m= 'm0' ]

%  electromagnetic vertex
  [ positron, electron, photon ; gpow = '1' ]
```

Suppose that `prop` represents the (name of the) 'propagator function', and `m` the 'propagator mass'. Coupling that description with the following code (which is assumed to be part of the diagram section of the style-file)

```
<propagator_loop>␣[prop](<momentum>,[m])*
<end>
```

leads, in the case of the diagram shown in Fig. 9, to the following output:

```
␣S(k1,me)*
␣P(k1-p1,m0)*
␣P(-k2,m0)*
␣S(k2+q1,me)*
␣S(k1+k2,me)*
```

This simple example shows how to obtain an output where each propagator type has its own name, and an appropriate mass as argument. As seen earlier, intrinsic style-keywords are 'select' identifiers[14] enclosed in angle brackets (the symbols '<' and '>', in fact). Now we can see that a function can give rise to a non-intrinsic style-keyword by enclosing its name in square brackets. This precludes any interference between the two types of keywords, so that any valid identifier can be used to name a function (though it is probably a good idea not to abuse that possibility).

It might be worth stressing that the domain of a function is a set of objects from the model-file, not a set of diagram embedded objects. For example, when the program outputs an image of `prop`, it looks only at the field names that appear in the propagator, not (eg) at the propagator-index; hence `[prop]` produces the same result for all the diagram propagators sharing exactly the same field names.

[14] Possibly ignoring a '`dual-`' prefix, that is.

Whenever appropriate definitions exist, both intrinsic and non-intrinsic keywords can appear in more than one kind of style-loop. Non-intrinsic keywords may also include the prefix '`dual-`' unless they derive from p-functions.

The following table shows which kind of functions are allowed in which type of loop (the presence of a full circle in a table entry denotes permission, and its absence interdiction). The strings `ff` and `vf` denote generic f-functions and v-functions, respectively; they map fields ϕ and vertices ν from the model into strings `ff`(ϕ) and `vf`(ν). The string `pf` denotes a generic p-function.

	i/o-loop	p-loop	r-loop	v-loop
`[ff]`	●	●	●	
`[dual-ff]`	●	●	●	
`[pf]`	●	●	●	
`[dual-pf]`				
`[vf]`	●	●	●	●
`[dual-vf]`		●		

The definitions of the generic keywords listed in the above table depend on the style-loop they appear in, but it is possible to present unified definitions using a few intrinsic loop-dependent keywords. Here they are:

- `[ff]` — the string `ff`(ϕ) where ϕ is the field that `<field>` refers to
- `[dual-ff]` — the string `ff`(ϕ) where ϕ is the field that `<dual-field>` refers to
- `[pf]` — the string `pf`(ϕ) where ϕ is either of the fields denoted by `<field>` and `<dual-field>`
- `[vf]` — the string `vf`(ν) where ν is the interaction vertex whose vertex-index is given by `<vertex_index>`
- `[dual-vf]` — the string `vf`(ν) where ν is the interaction vertex whose vertex-index is given by `<dual-vertex_index>`

Note that using `[dual-vf]` in the ray-loop would lead to problems. The issue is linked to the fact that some intrinsic keywords have abnormal definitions in this type of loop, and will hopefully be resolved in a later release.

31.2 Keywords that derive from constants

The constants declared in the model-file give rise to style-keywords too, and these may appear in any section of the style-file. For example, to the constant `model` corresponds the (non-intrinsic) keyword `[model]`.

32. Latest additions and extensions

32.1 The style-keywords `<loops1>` and `<loops2>`

As the `loops` statement can now set a range for the number of loops (or cycle-rank) of the generated diagrams, eg

```
loops = 2 thru 0 ;
```

it follows that the keyword `<loops>` may not produce a fixed output. The keywords `<loops1>` and `<loops2>`, which can be used in any section of the style-file, produce (respectively) the first and the second value that appear in the `loops` statement; if only one value is present then that is the value produced by either keyword.

32.2 The style-keyword `<index_offset>`

The keyword `<index_offset>` outputs the value declared in the `index_offset` statement, if there is one, else '`0`'. It may appear in any section of the style-file.

32.3 The style-keyword `<loop_momentum>`

This keyword may appear in any section of the style-file, and outputs the identifier declared in the `loop_momentum` statement; an error will occur if no such statement is present.

32.4 The style-keyword `<zero_momentum>`

This keyword outputs the identifier declared in the `zero_momentum` statement, if any, else the character '`0`'. It can be used in any section of the style-file.

32.5 More general `prologue` and `epilogue` sections

A restricted version of the style-loops `<in_loop>` and `<out_loop>` is now allowed in the prologue and epilogue sections. Naturally, in these sections the style-keywords allowed in those loops cannot depend on any diagram, hence some keywords have to be excluded (Section 33).

Those loops can be used to restate the basic definition of the input scattering process in a format recognized by the program (computer algebra system, say) that is going to process the output-file. Some additional information about the external fields, present in the model-file, can be produced as well. Specifically, the following information can be added or reformatted:

- the input process — ie the external fields and their momenta;
- other properties of the external fields — eg sign, index, type, values of f-functions and of p-functions.

There are a few more possibilities not related to fields, provided by some recently introduced style-keywords:

- the value(s) appearing in the `loops` and `index_offset` statements;
- the identifier(s) declared in the `loop_momentum` and in the `zero_momentum` statements.

Recall also that the constants of the input model can be used in any section of the style-file.

32.6 The style-keyword `<diagram_counter>`

The style-keyword `<diagram_counter>` outputs the number of diagrams generated in the current run, up to the moment when the respective output is to be produced.

The exceptional definition of `<diagram_index>` in the epilogue section will be cancelled with the next stable version, so that the output produced by this style-keyword is always the sum of the outputs of `<index_offset>` and `<diagram_counter>`. Each of these three keywords will then be able to be used in any section of the style-file.

32.7 The style-keywords `<local_symmetry_number>` and `<nonlocal_symmetry_number>`

The keyword `<symmetry_number>` produces a positive integer equal to the reciprocal of the symmetry factor. That integer can be written as a product of two others, however. Let $\texttt{Aut}(D)$ be the automorphism (ie ‘symmetry’, permutation) group of diagram D, and $\texttt{Aut}_p(D)$ the subgroup of $\texttt{Aut}(D)$ consisting of those automorphisms that leave every vertex fixed — that is, those symmetries that only involve permuting parallel propagators or ‘flipping’ self-loops. These are the kind of symmetries that may be called *local*.

The keyword `<local_symmetry_number>` produces the order of $\texttt{Aut}_p(D)$ and the other keyword, `<nonlocal_symmetry_number>`, the order of the quotient group $\texttt{Aut}(D)/\texttt{Aut}_p(D)$. Those two integers provide some basic information about the origin of the symmetry factor: if $|\texttt{Aut}_p(D)| > 1$ then (non-trivial) local symmetries exist; if $|\texttt{Aut}(D)/\texttt{Aut}_p(D)| > 1$ then there are (non-trivial) *non-local* symmetries. Those two cases may co-exist, obviously.

32.8 The style-keyword `<field_sub>` and its dual

The keyword `<field_sub>` (where *sub* stands for *subscript*) produces a variant of the field-index, only it is always positive. It may help simplify the generation of (eg) space-time indices for fields and momenta.[15] The set of values generated by this keyword is seldom a collection of consecutive natural numbers, but that should hardly be a problem.

Let r and s denote the number of incoming and of outgoing fields of the input process, and $t = \max(r, s{+}1)$. For an external field, `<field_sub>` produces the absolute value (ie the opposite) of the output of `<field_index>`. For an internal field, `<field_sub>` produces the result of the sum `2t + <field_index>`.

The dual keyword `<dual-field_sub>` can be used only in the propagator-loop and in the ray-loop (just like `<dual-field_index>`), and its output consists in the result of the sum `2t + <dual-field_index>` unless `<dual-field_index>` produces a zero, in which case it outputs a zero as well (this may occur only in the ray-loop). Thus, `<dual-field_sub>` may produce a zero in the ray-loop, but not in the propagator-loop.

The expected usage should probably be as follows: `<field_sub>` in the leg loops (in-loop and out-loop) and possibly in the ray-loop; `<field_sub>` and `<dual-field_sub>` in the propagator-loop, to ‘link’ the indices produced in the other loops.

Although it would have been possible to have a slightly more compact range for the output of `<field_sub>`, there appear to be two minor advantages as well, at least for debugging purposes. One, in every instance, the parity of the outputs of `<field_sub>` and of `<field_index>` is the same, and that also holds for their duals. Two, one can tell from

[15] Many users must have been doing something of the kind already, of course (as well as the kind of indexing enabled by the momentum-loop construct, described next).

the *set* of outputs of `<field_sub>` which fields are internal or external (and even incoming or outgoing), given that the subscripts of the internal fields (if any) form a special set of natural numbers — namely a set of even size consisting of those consecutive subscripts with the largest values, separated from the rest by a 'gap' (of size one or two). If there are no internal fields then either there is no such gap or there is a single number above the top gap. Anyhow, the identification of the external fields can then be based on the individual outputs of `<field_sub>` — this is similar to the corresponding identification based on the output(s) of `<field_index>` since it is just a matter of multiplying the output values by -1.

32.9 The style-keywords `<momentum_loop>`, `<momentum_term>` and its dual

The keyword `<momentum_loop>` sets a style-loop for the terms of the expression that would be produced by `<momentum>`, enabling a more complex output than the one provided by the latter keyword. The momentum-loop can be nested within either the ray-loop or the propagator-loop. The number of iterations is equal to the number of (nonzero, reduced) terms in the expression for the respective momentum if that momentum is nonzero, and is otherwise equal to 1.

The style-keyword `<momentum_term>` produces one term in each iteration, as if extracted sequentially from the output of `<momentum>`. Its dual `<dual-momentum_term>` also exists. In general, the keyword `<momentum>` can be replaced by the following construct.

```
<momentum_loop><momentum_term><end>
```

Now, each momentum term can be 'decorated'. The main role of the `zero_momentum` statement is to make the following kind of code

```
<momentum_loop><momentum_term>^{mu<field_sub>}<end>
```

produce consistent output, without generating dubious expressions like '`0^{mu2}`', for null momenta. For example, when the style-keywords `<momentum>` and `mu<field_sub>` produce (respectively) the expressions `k1-p1` and `7` then the latter loop construct should produce

```
k1^{mu7}-p1^{mu7}
```

Moreover, assuming that `k0` denotes the (declared) zero-momentum identifier then the same code should produce (eg)

```
k0^{mu4}
```

when a null momentum is found.

Thus, using the momentum-loop, it is possible to generate valid expressions in which any momentum term has a 'space-time' index. The typical $k^2 - m^2$ denominator can be constructed as well, though not in its simplest form. Although the initial size of the expressions for the amplitudes will tend to increase, the overall impact should be almost negligible unless the number of generated diagrams is quite large and the output of the program has to be stored in a file (and presently a new approach is emerging).

For now, `<field_sub>`, `<field_index>`, `<propagator_index>` and `<momentum_term>` are the only loop-dependent style-keywords allowed in the `<momentum_loop>`. Other style-keywords might be enabled later, should they be useful.

33. The list of intrinsic style-keywords for the prologue and the epilogue sections

The (extended) list of style-keywords for the `prologue` and `epilogue` sections:

```
<back>
<diagram_counter>
<diagram_index>
<dual-field>
<dual-momentum>
<end>
<field>
<field_index>
<field_sign>
<field_sub>
<field_type>
<full_time>
<in_index>
<in_loop>
<index_offset>
<leg_index>
<loop_momentum>
<loops1>
<loops2>
<momentum>
<out_index>
<out_loop>
<program>
<raw_time>
<statement_loop>
<statement_sub_loop>
<sub_statement>
<zero_momentum>
```

34. The list of intrinsic style-keywords for the diagram section

The keywords to be used when defining the style-loops:

```
<end>
<in_loop>
<momentum_loop>
<out_loop>
<propagator_loop>
<ray_loop>
<vertex_loop>
```

The keywords that do not depend on the style-loops:

```
<back>
<diagram_counter>
<diagram_index>
<index_offset>
<legs>
<legs_in>
<legs_out>
<local_symmetry_number>
<loop_momentum>
<loops>
<loops1>
<loops2>
<minus>
<new_elinks>
<new_loops>
<new_partition>
<new_topology>
<nonlocal_symmetry_number>
<propagators>
<sign>
<symmetry_factor>
<symmetry_number>
<vertices>
<zero_momentum>
```

The keywords that can be used only in a style-loop, but which do not have a dual:

```
<field_sign>                 <leg_index>
<field_type>                 <out_index>
<in_index>                   <propagator_index>
```

The keywords that can be used only in a style-loop, and which have a dual (at least sometimes):

`<field>`	`<dual-field>`
`<field_index>`	`<dual-field_index>`
`<field_sub>`	`<dual-field_sub>`
`<momentum>`	`<dual-momentum>`
`<momentum_term>`	`<dual-momentum_term>`
`<ray_index>`	`<dual-ray_index>`
`<vertex_degree>`	`<dual-vertex_degree>`
`<vertex_index>`	`<dual-vertex_index>`

Next, an updated table showing which (loop-dependent) keywords can be used inside which style-loops.

	`i_loop`	`o_loop`	`p_loop`	`v_loop`	`r_loop`	`m_loop`
`<dual-field>`	●	●	●		●	
`<dual-field_index>`			●		●	●
`<dual-field_sub>`			●		●	●
`<dual-momentum>`	●	●	●		●	
`<dual-momentum_term>`						●
`<dual-ray_index>`			●		●	
`<dual-vertex_degree>`			●		●	
`<dual-vertex_index>`			●		●	
`<field>`	●	●	●		●	
`<field_index>`	●	●	●		●	●
`<field_sign>`	●	●	●		●	
`<field_sub>`	●	●	●		●	●
`<field_type>`	●	●	●		●	
`<in_index>`	●					
`<leg_index>`		●				
`<momentum>`	●	●	●		●	
`<momentum_term>`						●
`<out_index>`		●				
`<propagator_index>`			●		●	●
`<ray_index>`	●	●	●		●	
`<vertex_degree>`	●	●	●	●	●	
`<vertex_index>`	●	●	●	●	●	

NB: Additional (loop-dependent) style-keywords may be allowed in the momentum-loop should that turn out to be useful.

Part V — Other topics

35. Command-line arguments

The command-line interface can be used to pass a few basic instructions to the program. The options described below include typically two leading hyphen-minus characters (eg `--version`), but it should be possible to use only one.

Some additional options that can be used in the conversion of input files to the latest specifications, for use with `qgraf-4`, are described in Section 41.

35.1 Defining the name of the control-file

For a normal run in auto-mode, the name of the control-file should be defined as a command-line argument, eg

```
qgraf a_control-file
```

In this example the program will try to read the file `a_control-file` located in the current directory. The restrictions described in Section 4.2 apply.

35.2 The option `--version`

Executing the command

```
qgraf --version
```

should result in the program outputting its own name and version (typically left-aligned on the standard output and terminated by a *newline*), and then exiting at once. No additional options nor arguments should be present.

For example, the character string produced by the latest version should be

```
qgraf-4.0.5
```

35.3 The option `--no_extra_chars`

With `qgraf-4`, some 'non-standard' characters (namely the `Tab` character and printable characters from certain 8-bit extensions of ASCII) should be tolerated in *annotations* (in any input file), if any such extension is supported by the compiler (and by the operating system). As there are multiple ASCII 8-bit extensions, of which at most one will be supported, it is not a very good idea to (eg) distribute model-files that include such characters — they should probably be used only 'privately' (the `Tab` is an exception, since it is a 7-bit character).

Nevertheless, if (whatever the reason) one wants to make sure that the input files contain no 'non-standard' characters, the command-line option `--no_extra_chars` enforces a run-time error whenever any such character is found.

36. The display-output

In what follows it will be assumed that the config option `noinfo` is not used, otherwise (ie typically) the display-output would be suppressed (Section 6).

36.1 The 'normal' display-output

Let us assume first that the program runs uneventfully (that is, issuing neither error messages nor 'alerts'). In that case, the first part of the display-output shows the version of the program and the statements found in the control-file. After that, though in `verbose` mode only, some basic information about the input model is displayed, eg

```
sectors:  1+ (P)   3 (V)
propagators:  (8)   2 (N+)  3 (C+)  1 (N-)  2 (C-)
vertices:  (17)   3^10  4^7
```

The first of those lines is displayed only if some sector is defined in the model-file. In this example the input (sub)model contains 1 propagator-type sector and 3 vertex-type sectors. If the `model` statement includes the name of a submodel then those two numbers refer to that submodel. A plus sign may follow each such number, indicating that there exist (respectively) propagator and vertex statements that are not in any sector block.

The subsequent line shows the number of propagators of the input (sub)model, as well as the respective subtotals per *propagator-type*, according to the following rules:

- the letter '`N`' refers to *'neutral'* propagators, ie those for which the particle is equal to the anti-particle;
- the letter '`C`' refers to *'charged'* propagators, ie those for which the particle and the anti-particle differ;
- the signs '`+`' and '`-`' denote whether the propagators are defined in terms of commuting or anti-commuting fields.

For example, the propagators of Dirac fermions (and of some ghost fields as well) contribute to the coefficient of `(C-)`, while the propagators of Majorana fermions contribute to the coefficient of `(N-)`. In the above example, the model-file defines a total of 8 propagators (5 'bosonic' and 3 'fermionic'); two of those bosonic propagators are neutral (string '`2 (N+)`') and the other three are charged (string '`3 (C+)`'); in the case of the fermionic propagators, one is neutral and two are charged.

Recent versions differentiate between 'true-propagators' and 'non-propagators': if a statement includes the keyword `external` then it counts as a 'non-propagator', otherwise as a 'true-propagator'. An extra line may then be displayed, eg

```
non-propagators:  (1)    1 (N+)
propagators:  (7)   1 (N+)  3 (C+)  1 (N-)  2 (C-)
```

Those numbers are mutually exclusive — in this second example the definition of the input (sub)model includes 8 propagator statements too.

The third displayed line (in the first example) shows the number of vertices of the input (sub)model and the respective subtotals per vertex degree (there is a total of 17 vertices, of which 10 are cubic and 7 are quartic).

What comes next (in either `info` or `verbose` display-mode) is the information about the number of connected diagrams for each compatible vertex-degree partition and (possibly) for each number of loops. An example follows, for which it is implicit that the respective control-file includes a statement of the form '`loops = 1 to 2 ;`'.

```
        #loops     v-degrees          #diagrams       subtotals


          1
                     -   4^1     ...       1

                   3^2   -        ...      2

                                                        3

          2
                     -   4^2     ...       2

                   3^2  4^1       ...      12

                   3^4   -       ...       6

                                                        20


             total =  23 connected diagrams
```

In this example, there are two possible partitions for 1-loop diagrams: `3^2` (diagrams with exactly two vertices, both cubic) and `4^1` (diagrams with exactly one vertex, but a quartic one). The corresponding numbers of diagrams are `2` and `1`, and the (1-loop) subtotal is `3`. Then, the same kind of information is shown for 2-loop diagrams.

Lastly, the total number of generated diagrams is shown (this part of the output can be slightly modified when a `count_to` statement is present). The `subtotals` column may be omitted when the additional information that would be provided is completely redundant (ie when the exact numbers that would be displayed already appear in the `#diagrams` column).

36.2 Obviating the diagram generation

Sometimes the program is able to detect that there are no diagrams, either for certain vertex-degree partitions or even for all such partitions, without having generated any diagram (work in progress). Those situations can be created by some combinations of options and/or statements. In such cases (those detected, that is) the display-output looks a bit different.

If the program detects a complete absence of diagrams (for instance, if the diagrams are required to be trees with self-loops), it will simply display

```
        total =  0 connected diagrams  **
```

without listing any partial result; otherwise, for each 'flagged' partition (ie a partition for which the program has determined that there can be no diagrams), the respective line shows only that partition, eg

```
           3^2  4^1
```

In either case, the modified display-output implies that the usual diagram generation was not attempted.

One other possibility of that kind consists in detecting whether the input process breaks some particle number conservation rule or, equivalently, some conserved charge — where *conserved* means (perturbatively) conserved in the input model, obviously. Still, for now, there is a separate program (Qgraf-r) which can be used for (eg) that purpose.

37. An application programming interface (API)

Let us say that QGRAF is in *auto-mode* when running autonomously (as it has been usually the case up to now), and in API-*mode* when being called from another (Fortran or C) program using the API described in this section. In API-mode, the control-file should include one or more consecutive `style` statements, eg

```
messages = 'm.txt' ;
style = 'f1.sty' ;
style = 'f2.sty' ;
style = 'f3.sty' ;
          ⋮
```

but no `output` statement. Thus, 'multiple-output' (or perhaps 'multiple-style') configurations are allowed too; the maximum allowed number of `style` statements is exactly the same in either mode (Section 9). If a message-file is not declared, any such information will be lost.

An *output-block* is a (possibly long) character string that consists in the output generated from the instructions included in some (single) section of one of the input style-files, executed exactly once; in particular, output-blocks may include *newlines.* In a typical run, one output-block will be produced for each prologue and epilogue section; in the case of the diagram section, one output-block will be built for each generated diagram and for each style-file. If a run produces n_d diagrams, and if there are n_s style-files, then that run may produce up to $n_s(n_d+2)$ non-empty output-blocks. The output-blocks are available in real-time to the calling program (ie the program that calls the API), which decides what to do with them.

37.1 QGRAF as a sub-program

An object file suitable for API-mode can be obtained from the same downloaded source file, though compiled with a different set of preprocessor-related options (Section 2.4). There is also an header file to be imported by C programs using the API. This interface consists mainly in a new module comprising some definitions and subroutines, including three interface subroutines (not to be used together), each of which providing a way to access in real-time some of the quantities computed by the program — by having their values written on the output-blocks. These interface subroutines (Section 2.4) may not only be employed to run QGRAF 'step-by-step' (eg diagram by diagram), treating it as a sub-program, but serve also as a kind of buffer that isolates the inner workings of QGRAF (which may continue to evolve independently) from the inner workings of any other computer program that may use QGRAF in that way. For simplicity, in what follows it will be assumed that there is a single such subroutine, fictitiously named `q_interface`.

Using the aforementioned interface consists essentially in calling repeatedly one of the interface subroutines. Each call should be of the form

```
call q_interface(sgnl1,sgnl2,ccs,csl,lint)
```

where `sgnl1`, `sgnl2` and `lint` are integer variables (the last one a 'long integer'), `csl` is an integer array (whose size should not be smaller than the number of input style-files), and `ccs` is a character string, long enough to hold the generated output-blocks (one per style-file), that is treated as an array of character strings. For example, the first call should tell QGRAF to initialize itself, read the control-file (whose filename is input in `ccs`) and the input files declared therein, storing any relevant information, and then return to the calling program.

The arguments `sgnl1` and (specially) `csl` may have to be set before each call is made. The value of `sgnl1` should be an *input-signal*, specifying what kind of computation should be performed. The array `csl` is 'multi-purpose': on entry, it can specify either the maximum lengths of the various output-blocks, or which output-blocks to construct; on return, it usually contains the lengths of the output-blocks generated during the call, if any such blocks were requested, although it may also contain the length of an error message returned in `ccs` or the number of generated diagrams. Occasionally, `ccs` and `lint` are used as input arguments.

The arguments `sgnl2`, (not often) `lint`, and (very often) `ccs`, `csl` are set by QGRAF and returned to the calling (sub-)program; they should be explicit (over-writeable) variables and arrays. The value returned in `sgnl2` is a *return-signal*, which provides some feedback on the call's success, or lack thereof. When a call requests some output-block(s), and unless an exception occurs, the (returned) arrays `ccs` and `csl` should contain those output-blocks and their respective lengths (one per entry, following the sequence in which the respective style-files were declared).

37.2 Basic input-signals

The input-signals tell QGRAF what kind of computation to perform (of which there exist only a few pre-defined types). The following list should include the most common.

- `qisgnl%init`

The signal that instructs QGRAF to 'start-up' and then read (and parse) the input files; it should be the input-signal of the first call.

- `qisgnl%count`

Requests the total number of (connected) diagrams that match the input conditions, if `lint` is set to zero (on input). A potentially long call could then take place, but a 'bounded' call (similar to what can be achieved with a `count_to` statement) can be requested by setting `lint` as a positive integer.

- `qisgnl%msg_count`

This signal requests the number of messages suppressed up to that point (since the last initialization).

- `qisgnl%prologue`

Requests the output-block(s) for the prologue section of each style-file.

- `qisgnl%diagram`

The signal that requests the output-block(s) for the diagram generated during the call (ie for the 'next' diagram, if any, even for the first diagram).

- `qisgnl%epilogue`

Requests the output-block(s) for the epilogue section of each style-file.

- `qisgnl%stop`

Instructs QGRAF to close any open file (the message-file, typically) and terminate the current job; the option `flush` is implied. It can be used at any time after initialization, even if the diagram generation is not finished.

On entry, and when some output-blocks are requested, each of the entries of `csl` should also be an input-signal, though one of a slightly different kind. The next two signals (which can be interpreted as *'yes'* and *'no'*) are the only values allowed.

- `qisgnl%y`, `qisgnl%n`

  The n^{th} output-block (ie of the type defined by `sgnl1`, and derived from the n^{th} style-file declared in the control-file) is computed if and only if `csl(n)` equals `qisgnl%y`.

37.3 Return-signals

A *return-signal* tells the calling program whether or not the call was successful and possibly what kind of error occurred. There are two distinct return-signals, given next, that can be produced in a non-eventful run (that is, when none of the exceptions described further below occurs).

- `qrsgnl%ok`

  The signal returned when the call was successful. If some output (other than `sgnl2`) ought to have been produced then the contents of `ccs` and `csl` should be valid (unless there is a bug, of course).

- `qrsgnl%end`

  This return-signal means that there are no more output-blocks of the type requested — or, if returned on the first call of its type, that there are no such output-blocks at all (the latter situation should not happen except for a call with `qisgnl%diagram`).

There are also three 'error signals', defined below, to let the calling program know not only that some run-time exception occurred but also the generic type of that exception. If there is a corresponding error message then `csl(1)` should be positive, the text of that message should be stored in `ccs(1)`, and its length should be equal to `csl(1)` — excluding any trailing control character(s). Only the messages for (detected) run-time errors are passed on to the calling program (at most one message per run, obviously). The warning messages should be saved in the message-file, once (and if) it has been opened.

- `qrsgnl%input_error`

  This is the signal returned when QGRAF detects an error in some input file or in some call to the interface subroutine.

- `qrsgnl%q_error`

  The signal returned when QGRAF detects some inconsistency with its current state, likely due to a problem with its own code.

- `qrsgnl%os_error`

  The signal returned when QGRAF experiences any type of I/O problem.[16]

These return-signals should occur rarely. Once problems like lack of disk space and wrong file permissions are excluded, finding a `qrsgnl%os_error` signal should be a very rare event since it will likely mean that a serious software or hardware error has occurred.

[16] This means input/output problem, and includes reading from files and writing to files.

37.4 Some examples

For interoperability's sake, and also to avoid some duplication, the types of the variables shared by (or with) QGRAF are interoperable C-types — ie (*i*) `c_int32_t` for `sgnl1`, `sgnl2` and `csl`, (*ii*) `c_char` for `ccs`, and (*iii*) `c_int64_t` for `lint` — even when the calling program is a Fortran one. In C, the corresponding types are `int32_t`, `char`, and `int64_t`, obviously. It should be noted that it is not that difficult to generate a set of diagrams whose cardinality is not representable with `int32_t` — currently, a few hours of CPU time will be sufficient, at least if only their count is required. Still, the type chosen for `lint` should preclude the need for future adjustments.

The following lines of code show how to initialize QGRAF, using Fortran; the ancillary files `pxq.c` and `pxq08.c` include a C version of most of the examples given below.

```
sgnl1= qisgnl%init
ccs(1:121)= "'qgraf.dat'" // c_null_char
csl(:)= int(4096,c_int64_t)
call q_interface(sgnl1,sgnl2,ccs,csl,lint)
```

The encoded name of the control-file, terminated by a `c_null_char` character, should fit into the first 120 positions of `ccs`. Space characters at either end (ie not in the sub-string delimited by the single quotes), if any, are automatically deleted; other spaces can be deleted with the config option `noblanks`.

In that call, the entries of `csl` specify the maximum allowed lengths for the output-blocks to be derived from the style-files (one maximum per style-file). In this example, all of those lengths are equal to 4 096. More generally, and at present, each (declared) maximum should be comprised between 128 and 16 384 (ie 2^7 and 2^{14}), and their sum should not exceed (approx.) 46 000. In each case, the *effective* maximum will be equal to the declared value *minus* 8. The reserved positions exist as a very minimal precaution against potential 'overflows', and also to allow for one or two control characters. Those values will remain fixed until the subsequent re-initialization.

Still in the same type of call, if `qrsgnl%ok` is returned in `sgnl2` then the following information should be returned too: the version of QGRAF (in `ccs`, starting at position 1 and terminated by a `null` character) and the corresponding string length (in `csl(1)`).

Recall that the name `q_interface` is not real. The actual names of the interface subroutines are given in (Section 2.4).

When an exception occurs, the error message is returned in `ccs`, its length in `csl(1)`. A re-initialization should still be allowed if the return-signal is `qrsgnl%input_error`. In every other case, any subsequent call is returned at once with the original error signal and the run will probably have to be abandoned.

Every call to `q_interface` should be followed at once by a check on `sgnl2`, for example

```
if( ( sgnl2 .ne. qrsgnl%ok ) .and. ( sgnl2 .ne. qrsgnl%end ) )then
   call my_error_checking(sgnl1,sgnl2,ccs,csl)
end if
```

where subroutine `my_error_checking` should decide what to do in case of error.[17] The body of that subroutine might have the following structure.

[17] Possibly notify the job owner and then stop...

```
if( sgnl2 == qrsgnl%os_error )then
   (...)
else if( sgnl2 == qrsgnl%input_error )then
   (...)
else if( sgnl2 == qrsgnl%q_error )then
   (...)
end if
```

To do a *'counting run'* one may proceed as follows.

```
sgnl1= qisgnl%count
lint= 0
call q_interface(sgnl1,sgnl2,ccs,csl,lint)
```

Setting `lint` to zero makes the call 'unbounded' (ie the program will try to generate every possible diagram), but setting `lint` as a positive integer would automatically make the call 'bounded', with `lint` becoming the upper bound for the number of diagrams to be constructed (this would be the equivalent of having a `count_to` statement in auto-mode). The number of (actual) generated diagrams should be returned in `lint`. This kind of call should be made before requesting any output-block for either the diagram section or the epilogue; if it is made, QGRAF will have to be re-initialized (possibly using the same control-file) before producing any output-block for the diagram section.

Once re-initialized, QGRAF should once again be able to construct any requested output-block(s). In the case of the diagram section, there should be a call per diagram — until the signal `qrsgnl%end` is returned. Meanwhile, any requested set of output-blocks will be returned in `ccs` as if this string had been divided into sub-strings with fixed lengths, specified in the previous initialization; the actual lengths of the output-blocks will be returned in `csl`. The following lines of code show how to request (all) the output-blocks for the prologue(s).

```
sgnl1= qisgnl%prologue
csl(:)= qisgnl%y
call q_interface(sgnl1,sgnl2,ccs,csl,lint)
```

To obtain the output-blocks for the diagram section, one might do the following.

```
sgnl2= qrsgnl%ok
do while( sgnl2 == qrsgnl%ok )
   sgnl1= qisgnl%diagram
   csl(:)= qisgnl%y
   call q_interface(sgnl1,sgnl2,ccs,csl,lint)
   if( sgnl2 == qrsgnl%ok )then
      (...)
   else if( sgnl2 == qrsgnl%end )then
      (...)
   else
      call my_error_checking(sgnl1,sgnl2,ccs,csl)
   end if
end do
```

The first case of that `if` block corresponds to a successful call (ie a new diagram has been generated), and the respective code should describe what to do with each output-block. The second case corresponds to the natural end of the current diagram generation job.

Not all three sections of the style-file have to be made use of, but the usual relative precedence should be observed. For example, the output-blocks for the epilogue can be requested (once) at any time but then it is not possible to request output-blocks for any of the other sections without re-initializing.

The next example shows how to obtain the number of warning messages suppressed up to that point, whether or not saved in the message-file.

```
sgnl1= qisgnl%msg_count
call q_interface(sgnl1,sgnl2,ccs,csl,lint)
```

That number should be returned in `lint`.

One last example shows how to instruct QGRAF to close any open files(s). If no exception occurs (and if there are no bugs), only the message-file may be open when that call is made.

```
sgnl1= qisgnl%stop
call q_interface(sgnl1,sgnl2,ccs,csl,lint)
```

The number of suppressed warning messages (since the previous initialization) is returned in `lint`. After returning, QGRAF will wait for the next re-initialization, if any. As always, it is necessary to check whether some error occurred.

37.5 Advanced input-signals

A second set of input-signals (to be implemented, except `qisgnl%hold`) allows for a more efficient diagram searching, assuming that it is possible to utilize QGRAF's algorithmic structure — ie the fact that the diagram generation involves four hierarchical levels (cycle-rank, vertex-degree partition, unlabelled topology and labelled topology). These signals should not be needed in most situations — they are intended for demanding, CPU-intensive cases, with unusual selection criteria.

Suppose one is interested in finding a set of rare diagrams (as a fraction of the total) that satisfy some special criteria not implemented in QGRAF. Then, if the total number of diagrams to be constructed is potentially huge, it would be very convenient to avoid generating as many irrelevant diagrams as possible. For example, if the topology of the current diagram cannot be used to generate a relevant diagram, one would want to skip every other diagram with that topology. That is the basic idea that lies behind the following signals, which may be chosen at will — one at a time, of course (see also Section 30.5). Still, the only signal that can set the diagram generation in motion is `qisgnl%diagram`.

- `qisgnl%elinks`

This signal instructs QGRAF to ignore any further diagram with a labelled topology[18] equal to that of the current diagram, and thus to produce the first subsequent diagram for which `<new_elinks>` produces a '`1`'.

[18] This refers not only to the underlying graph but also to the external field conguration.

- `qisgnl%topology`

  This signal is similar to the previous one, except that it refers to the unlabelled topology (hence the 'jump' can be longer). It request the first subsequent diagram for which `<new_topology>` produces a '`1`'.

- `qisgnl%partition`

  In this case the next diagram with a new vertex-degree partition is requested.

- `qisgnl%loops`

  This signal requests the next diagram with a new cycle-rank (ie number of loops).

Each of the last four signals can be used with nearly every call — that is, after the diagram generation has been initialized, and before `qrsgnl%end` is returned. One additional signal exists, as detailed next.

- `qisgnl%hold`

  This input-signal tells QGRAF to construct additional output-blocks for the latest generated diagram — in accordance with the (new) values of the entries of `csl` — instead of trying to find a subsequent diagram.

This last signal may sometimes be used to reduce the (total) number of output-blocks to be constructed since it allows the calling program to make more than one call for the same diagram, with different (typically 'orthogonal') `csl` arrays. For example, if the selection criteria can be applied without generating all the output-blocks for the current diagram, it can be more efficient to delay the construction of some output-block(s) until that diagram is found to satisfy the required criteria (and to skip their construction otherwise).

37.6 Further details

In API-mode, there are some constraints on the control-file: neither the statements

```
count_to
output
```

nor the config options related to the display-mode and (strictly) to output-files, ie

```
info
lf
noinfo
nolist
verbose
```

are available; moreover, the following options should not appear in the `options` statement if any of the advanced input signals is to be used (`qisgnl%hold` excepted).

```
topol
new_elinks
new_topology
new_partition
new_loops
```

In some calls, additional information is returned for cross-checking purposes. On an initialization call, the number of style-files is returned in `lint` (in principle, that number is known to the calling program since the control-file must be constructed in advance). On a call that returns `qrsgnl_end`, the number of generated diagrams is returned in `lint`.

The output-blocks may or may not contain 'newlines', depending on what the style-files specify. QGRAF can construct long output-blocks without such characters. A section of the style-file may consist of many lines, but inserting `<back>` at the start of each line (except the first) leaves only the terminating newline. In API-mode, however, trailing newlines are automatically erased, and a single `null` character is added for compatibility with C (the `null` character does not contribute to the lengths returned in `csl`). Deleting any non-trailing newlines can be achieved with the config option `no_ntnls`.

37.7 The ancillary files

The latest 'pack' includes (in directory `api_mode`) three examples of interfacing, one written in Fortran and two in C, respectively

```
pf08.f08
pxq08.c
pxq.c
```

Hopefully, some insight will be gained by perusing their code and utilizing (one of) them as template(s). All of those examples use distinct interface subroutines, that is, each of the three currently available interface subroutines is used by one of those programs. It is very important to pick the right subroutine for the calling program, of course; Section 2.4 includes further details. C programs should include the header file `qgraf.h` by means of a preprocessor directive, eg

```
#include "qgraf.h"
```

As already mentioned, use is now made of (a few, simple) preprocessor directives and of the GNU preprocessor. With the help of the `make` command, executed in the directory that contains the `Makefile`, a binary (that also includes QGRAF) can be created for each of the above source files, eg

```
make pxq
```

Once created, they can be run on a terminal, for instance

```
./pxq
```

38. The diagram sign

The *diagram sign* has been perhaps QGRAF's least understood feature. This state of affairs seems to derive mainly from the program's ignorance about (or rather, avoidance of) *graphical rules* — the most well known being *'for each fermion loop, multiply the amplitude by* -1*'*, of course. Then, what does QGRAF *do*?

The explanation for there being a possible relative minus sign between two distinct diagrams (for the same scattering process) follows directly from Wick's theorem in the presence of anti-commuting fields, and that is precisely the approach implemented in the program. To compute the sign of a Feynman diagram D (hereafter, we will naturally assume that D depends on some anti-commuting fields), the program starts by placing side by side (as a product) the vertices into which D can be decomposed. For example,

$$F_1 = (\bar{\Psi}_2\,\Psi_{-1}\,A_3)\,(\bar{\Psi}_{-2}\,\Psi_7\,A_5)\,(\bar{\Psi}_{10}\,\Psi_1\,A_6)\,(\bar{\Psi}_8\,\Psi_9\,A_4)$$

is the vertex product for the diagram shown in Fig. 14. The field ordering in each vertex is assumed to coincide with that of the respective vertex statement given in the model-file; the subscripts, which match the labels in that diagram, are the field-indices computed by the program. Those vertices have been reduced to a product of 'plain' fields, as the other contributions (coefficients, space-time indices, and so on) are not needed here.

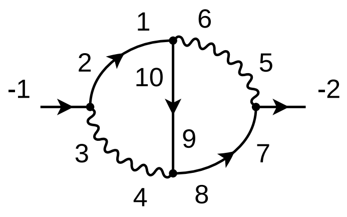

Fig. 14. A diagram (in QED) and its field-index numbering.

In F_1, the vertex sequential order is irrelevant because each vertex is (for this purpose) a commuting quantity, as it comprises an even number of anti-commuting fields. In contrast, the relative ordering of the anti-commuting fields in each vertex is clearly relevant. In any case, one has to choose an (arbitrary) initial vertex ordering to do the computation, and after that choice is made no vertices will be permuted — only a sequence of simple transpositions, each one involving exactly two fields, will be performed. If one discards the commuting fields, which clearly play no role in the present computation, F_1 will be reduced to the ordered product

$$F_2 = \bar{\Psi}_2\,\Psi_{-1}\,\bar{\Psi}_{-2}\,\Psi_7\,\bar{\Psi}_{10}\,\Psi_1\,\bar{\Psi}_8\,\Psi_9$$

which may also be regarded as a sequence. The next point to consider is that, whatever the type of field, the propagator ordering chosen by the program is always of the form

$$\langle\,\Phi_{2k-1}\,\Phi_{2k}\,\rangle$$

where the subscripts denote (internally generated) field indices. Thus, in our example, the internal anti-commuting fields will be paired into the following sub-sequences

$$\langle\,\Psi_1\,\bar{\Psi}_2\,\rangle,\ \langle\,\Psi_7\,\bar{\Psi}_8\,\rangle,\ \langle\,\Psi_9\,\bar{\Psi}_{10}\,\rangle.$$

For explanatory purposes, it is useful to constrain the field transpositions allowed in this part of the computation; without loss of generality, we shall require that the propagator pairing be achieved without changing the relative ordering of the external fields. With this restriction, once F_1 is fixed the parity of the propagator pairing operation will be well defined for each diagram. If this step takes t_p transpositions, the contribution to the sign will be $(-1)^{t_p}$. The internal fields are then deleted, as they are no longer needed. In our example, since an odd number of transpositions is required to transform F_2 into the sequence

$$F_3 = \Psi_{-1}\ \bar{\Psi}_{-2}\ \Psi_1\ \bar{\Psi}_2\ \Psi_7\ \bar{\Psi}_8\ \Psi_9\ \bar{\Psi}_{10}$$

we obtain a factor equal to -1.

The last part of the computation involves the external fields. Here, QGRAF assigns (arbitrarily) a positive sign to its generic 'reference' sequence, namely

$$\Phi^{out}_{-2}\ \Phi^{out}_{-4}\ \dots\ \Phi^{out}_{-2s}\ \Phi^{in}_{-2r+1}\ \dots\ \Phi^{in}_{-3}\ \Phi^{in}_{-1},$$

where r and s denote the number of incoming and of outgoing fields, and the subscripts are the field-indices (which are negative, since the fields are external). Any occurring sequence of external fields obtained in the previous step is then compared with the reference sequence (divested of its commuting fields), and the parity of the number of transpositions t_e needed to convert one sequence into the other is determined. Hence a new factor $(-1)^{t_e}$ is generated, and the diagram sign is then defined as $(-1)^{t_p+t_e}$.

In our example, once the internal fields are dropped from F_3 we are left with the sequence

$$F_4 = \Psi_{-1}\ \bar{\Psi}_{-2}$$

which can be converted into the reference sequence using a single transposition. Therefore, as $(-1)^{t_p} = (-1)^{t_e} = -1$, QGRAF produces the sign `+` for that diagram.

In comparison with a graphical rule, the preceding definition of diagram sign might seem (to a human, and concerning evaluations 'by hand') somewhat complex, even error-prone; nevertheless, it is closely related to Wick's theorem, and there is no difficulty whatever in programming it. Still, users who habitually rely on graphical rules might find some of the consequences of that definition rather unexpected. For that reason, let us take a look at a number of problems that may arise out of some misunderstanding or mere inattention.

38.1 Problem 1

Suppose we have defined QED as follows,

```
%  propagators
  [ photon, photon, +1 ]
  [ positron, electron, -1 ]

%  vertex
  [ positron, electron, photon ]
```

where `electron` corresponds to a Dirac field Ψ and `positron` to $\bar{\Psi}$. The usual statement for that fermionic propagator should be

```
[ electron, positron, -1 ]
```

as this is the one that corresponds to the ordered contraction $\langle \Psi \bar{\Psi} \rangle$. The reason why the former statement can be a problem is that QGRAF will then sort the internal anti-commuting fields in a different way, and may thus compute a different sign. Then, if each propagator expression in the program's output is replaced by the usual Feynman rule, the result can be an incorrect amplitude. Declaring the 'wrong' vertex may lead to similar problems.

In QED, as defined above, using the conventional propagator expression does not always lead to problems, due to the following: for a fixed scattering process, and a fixed order of perturbation theory, every diagram has the same number of fermionic propagators; moreover, the number of fermionic propagators in n-loop diagrams and in $(n+1)$-loop diagrams differs by *two*. Nonetheless, there exist many models for which one can find two diagrams (for the same process) with the same sign if the fermionic propagators are declared in one way, and opposite sign in the other.

38.2 Problem 2

Suppose we want to compute the amplitude of some scattering process that involves fermions (in the initial and/or final state), both at leading order and at some higher orders. We declare

```
in = electron[p1], positron[p2] ;
out = photon[q1], Z[q2] ;
loops = 0 ;
```

for the leading order, and

```
in = positron[p2], electron[p1] ;
out = photon[q1], Z[q2] ;
loops = 1 ;
```

for the next-to-leading order. If the two initial fermion states are permuted, the parity $(-1)^{t_e}$ changes (for *every* diagram). Oops, the next-to-leading order amplitude is to be added to the leading order amplitude, but now every relative sign between a 0-loop diagram and a 1-loop diagram will be wrong.

38.3 Problem 3

We have a model with both Dirac and Majorana fermions, say. Then we choose the scattering process and the order of perturbation theory, and have QGRAF write down the corresponding 'amplitude'. Now we pick the (commuting) expression for that amplitude, which includes the diagram signs computed by the program, and decide to 're-orient' some fermionic propagators in that expression. For example, we make some substitutions such as

`prop( Psi(3,k1), Psi(6,-k1) )` $\rightarrow$ `prop( Psi(6,-k1), Psi(3,k1) )`

where either of these expressions denotes a propagator function for a Majorana fermion `Psi`. That can also be problematic since those signs were computed by assuming a certain field ordering (eg for the propagator fields). If the redefined propagators had been used instead then different signs might have been produced for some diagrams.

We might decide also to exchange some arguments describing that same fermion, this time in some sub-expressions representing vertex (Feynman) rules. For instance, we might have an amplitude that includes a sub-expression such as

```
vertex( Psi(5,k1), Psi(2,-k1-k2), Phi(7,k2) )
```

where `Phi` is a bosonic field, and then we (simply) exchange the arguments `Psi(5,k1)` and `Psi(2,-k1-k2)`. Since those fields are identical there is no problem, right? Not really, that exchange would have affected the computation of $(-1)^{t_p}$.

38.4 Problem 4

The Path Integral formulation of Quantum Mechanics 'says' that the amplitude for a given scattering process should include the contribution of every 'path' leading from the initial to the final state (in the corresponding interpretation, that is why one adds the contributions of multiple Feynman diagrams, of course). Let us consider an experiment that has various possible outcomes, and for which — as a result of the measuring apparatus not being precise enough, say — some of the outcomes are not distinguishable from one another. Then, the amplitude for a certain (measurable) final state can be obtained by adding the amplitudes for the individual processes compatible with that measurement. In this case, the relative sign between diagrams for (a priori) distinct processes becomes critical.

The emission of low energy photons in a high energy collider provides a simple example. Experimentally, a scattering process like

```
in = electron[p1], positron[p2] ;
out = muon_minus[q1], muon_plus[q2] ;
```

may often be indistinguishable from (eg)

```
in = electron[p1], positron[p2] ;
out = muon_minus[q1], muon_plus[q2], photon[q3] ;
```

specially for small (spacial) momentum $|q_3|$. Here, QGRAF seems to compute the correct relative sign between the (former) 'non-radiative' process and the (latter) 'radiative' process, *if* the respective statements match as above (one should check all the same, of course). Nevertheless, since QGRAF does not address that type of problem, other means must be employed to determine an appropriate global sign for each process. Moreover, any such adjustments should preferably be (determined and) made at an early stage, to avoid having to redo as many runs as possible. The sign of an amplitude may often be adjusted by permuting two incoming (or outgoing) fermions, but that is a minor point.

38.5 Solutions

It is always possible to ignore the sign computed by the program (by omitting any reference to it in the style-file), although that will require the user to implement some substitute definition (completely, that is). That seems the best option if one wishes to implement some sign convention not supported by the program.

Alternatively, if one does not care much about which sign convention is actually used, but wants nonetheless to be able to perform some substitutions that correspond in practice to fermionic exchanges, there is another (possibly easier) type of solution which consists in fixing the problematic substitutions (instead of discarding QGRAF's sign altogether). The basic idea is that any substitution involving an odd number of fermionic transpositions should also generate a compensating factor equal to -1.

Solutions to Problem 1

There is more than one fix, apart from just using the conventional propagator declaration. For instance, one may use (at the symbolic processing stage) a modified Feynman rule for the propagator, differing from the standard one by a factor -1. Alternatively, the model could be redefined by letting Ψ describe the `positron` and $\bar{\Psi}$ describe the `electron`. This would have to be reflected on the form of the electromagnetic vertex, and one should also have to consider what would the electric charge constant e define. Similar comments apply to the non-conventional vertex declaration. Innovation of this kind is seldom a good idea, as the task of comparing one's results with existing results can (very likely) become more difficult; code debugging may also be harder.

Solution to Problem 2

Only the loop order of the diagrams should be changed, obviously. The extended `loops` statement might help, but the diagrams will then be listed in the same output-file (hence they will have to be selected afterwards, depending on the order of perturbation theory being computed).

Solution to Problem 3

For each fermionic transposition, whether in a propagator or in a vertex, there should be an additional -1 factor. That type of problem may also occur as a result of using a wrong substitution rule. Additionally, as $(-1)^2 = 1$, it may sometimes happen that *'two wrongs make a right'* (and that also applies to the other problems).

38.6 Further comments

Graphical rules were invented at a time when Feynman diagrams were generated by hand, and they allowed the person(s) doing some QFT calculation to easily (ie 'visually') determine a practical diagram sign. Nonetheless, while graphical rules for QED and QCD are quite simple, they are more complex for other types of models — pure graphical rules may not even exist. The approach used by QGRAF can be employed in general, irrespective of the vertex degrees and numbers of anti-commuting fields, although the output may have to be processed somewhat to obtain expressions of the desired form.

Nowadays, with the use of automatic set-ups being the norm, graphical rules are becoming less relevant. In addition, 'simplifying' Wick's theorem consists in practice in adopting some (further) convention — while one of the guidelines behind QGRAF has been the adoption of as few conventions as possible, specially when they are not general enough.

A continued reliance on graphical rules may also induce the belief that the diagram sign is something much more rigid than it really is. By that, we mean that the sign derived from any fixed set of graphical rules depends in fact on some convention(s) and/or assumption(s), which those rules may hide. Lastly, graphical rules do not necessarily solve Problem 4 either.

39. Models with explicit propagator mixing

For diagram generation purposes, a model features explicit propagator mixing if there is at least one field appearing in two (or more) propagator statements. Although this type of model is not accepted (at least not yet), there is a way to obtain the corresponding Feynman diagrams — namely, by replacing the original model by an appropriately transformed model, as described in the following paper.

Feynman graph generation and propagator mixing, I
Comput. Phys. Commun. 269 (2021) 108103.
`https://doi.org/10.1016/j.cpc.2021.108103`

In general terms, that type of transformation can be described by an algorithm that involves

- introducing new (ancillary) 'charged' fields;
- modifying the original set of propagators, to eliminate explicit mixing;
- adding new interaction terms, similar to the existing ones, but depending on the new fields as well.

Although (as originally described) that algorithm applies directly to models whose propagators do not contribute to the order of perturbation theory (eg their Feynman rules are independent of the coupling constants), it should be possible to apply it to many models in which some of the propagators do contribute — namely, by defining appropriate weights (here, p-functions) in the model-file and then using `psum` statements.

It is perhaps advisable to let the input process (specified by the incoming and outgoing fields) be defined in terms of the fields of the original (non-transformed) model only, keeping the ancillary fields as internal (ie appearing only in propagators). The reason is twofold: clearly, ancillary external fields are not needed for studying the original model, and using those fields adds an extra complication, to be described below, which may lead to errors.

Let us consider a Lagrangian density $\mathcal{L}(\phi_1, \phi_2)$ that depends on two self-conjugate (real) fields ϕ_1 and ϕ_2, and whose quadratic part includes a mixing term involving both fields, eg

$$\mathcal{L} = \tfrac{1}{2}\partial_\mu\phi_1\partial^\mu\phi_1 + \tfrac{1}{2}\partial_\mu\phi_2\partial^\mu\phi_2 + a\partial_\mu\phi_1\partial^\mu\phi_2 + \ldots$$

In this case, as shown in Example 4.1 of the above mentioned paper, the transformed model depends also on a conjugate pair of ancillary fields, to be denoted by ϕ_3 and ϕ_4. To convert the diagrams obtained for the transformed model into the diagrams of the original model, the *internal* ancillary fields ϕ_3 and ϕ_4 should be replaced as follows:

$$\phi_3 \to \phi_1, \qquad \phi_4 \to \phi_2 .$$

In that paper, there is no discussion concerning the distinction between incoming and outgoing fields. Implicitly, the definition of u-product entails that every external field is an incoming field (the opposite convention is allowed, but it is perhaps not as common). As Qgraf requires the incoming and the outgoing fields to be properly identified, however, one may find that the previous substitution rules do not necessarily apply to external fields. In fact, while they are still valid for incoming fields, for outgoing fields one has instead

$$\phi_3 \to \phi_2, \qquad \phi_4 \to \phi_1 .$$

For example, since ϕ_3 and ϕ_4 constitute a conjugate pair, the input process

```
in = phi1[p1] ;
out = phi3[q1] ;
```

is ‘equivalent’ to

```
in = phi1[p1], phi4[p2] ;
out = ;
```

which (by the former substitution rules) is ‘equivalent’ to

```
in = phi1[p1], phi2[p2] ;
out = ;
```

which (as ϕ_2 is self-conjugate) is ‘equivalent’ to

```
in = phi1[p1] ;
out = phi2[q1] ;
```

which should be compared with the initial process. These processes are *essentially equivalent*, in the sense that the set of diagrams generated for each process can be easily transformed into the set of diagrams for any other equivalent process; in particular, those sets have the same number of diagrams. In contrast, the scattering process

```
in = phi1[p1] ;
out = phi4[q1] ;
```

is not ‘equivalent’ to the previous ones.

Part VI — Additional information

40. Automatic downloads, licensing

It might be useful to have a simple way of automatically downloading a patched version instead of an outdated version with known problems, or even download a new (minor) version instead of the previous one. The following version-linking set-up has been implemented.

- any future release will be numbered '`x.y.z`', that is, using major, minor, and patch numbers (for instance, the first version of '`qgraf-4`' should be `qgraf-4.0.0`);
- a request to download `links/qgraf-x.y.tgz` (with valid `x` and `y`) is converted into a request for the version `qgraf-x.y.z.tgz` with the highest (available) `z`; the end result should be the latest `qgraf-x.y.z` for the given `x` and `y`.
- a request to download `links/qgraf-x.tgz` (with valid `x`) is converted first into a request for the version `links/qgraf-x.y.tgz` with the highest `y`, which will then be converted by the previous rule;
- a request to download `links/qgraf-x.s.tgz` (with integer `x` and literal `s`) is converted into a request for the version `links/qgraf-x.y.tgz` with the highest `y` such that `qgraf-x.y` has been declared *'stable'*;
- other requests, including those for specific versions, will not be converted (irrespective of availability);
- automatic downloading requests should now refer to the directory `links` eg
```
wget --quiet --user=anonymous --password=anonymous -O ./qgraf.tgz \
   http://qgraf.tecnico.ulisboa.pt/links/qgraf-4.0.tgz
```
- after each new release, there will be a 'grace period' (to be announced, possibly variable) before the corresponding conversions become effective; that would be coupled with an optional, automatic message system (see below).

This should not be contentious since it is optional and does not remove existing features. An *'alert system'*, which would allow 'package owners' and other users to automatically receive news and (above all) alerts about the program, is being considered.

Please do not implement or describe openly the (ie any) downloading method in some webpage, ready for 'bot use' (alas, there seems to be more than enough bot generated web traffic already). The information here provided is for 'package-scripting' only (ie for those packages that rely on this program).

Some versions that are no longer referenced on the website's *Downloads* page might still be temporarily available in the above mentioned `links` directory. At the time of the release of `qgraf-4.0.5` the convertible requests and their targets are as follows.

`links/qgraf-3.4.tgz`	→	`qgraf-3.4.2.tgz`
`links/qgraf-3.5.tgz`	→	`qgraf-3.5.3.tgz`
`links/qgraf-3.6.tgz`	→	`qgraf-3.6.10.tgz`
`links/qgraf-3.tgz`	→	`qgraf-3.6.tgz`
`links/qgraf-3.s.tgz`	→	`qgraf-3.6.tgz`
`links/qgraf-4.0.tgz`	→	`qgraf-4.0.5.tgz`
`links/qgraf-4.tgz`	→	`qgraf-4.0.tgz`

and the non-convertible requests are

```
links/qgraf-3.4.2.tgz
links/qgraf-3.5.3.tgz
links/qgraf-3.6.9.tgz
links/qgraf-3.6.10.tgz
links/qgraf-4.0.5.tgz
```

40.1 Stable versions and licensing

A *stable version* is a version that should be expected to be available and supported for a reasonably long period (five years, at least), within my ability and resources, and assuming that the present circumstances do not worsen in any relevant way, of course. Note that by *version* I mean something like `qgraf-x.y` where `x` and `y` denote the major and the minor version numbers. Since declaring a version as stable the moment it is released does not seem to be the best strategy, there will be a delay (of at least five months) so that (*i*) a better perspective on the program development may be obtained and (*ii*) there is an additional opportunity to find and eliminate any remaining bugs, or even to refine or add some feature. Once a version is declared stable, it should not be modified except for fixing some anomaly or unexpected behaviour, and even then in a minimal way; that includes both actual errors and potential errors, ie code that is already known to be erroneous and code thought likely to lead to problems.

Not every version will be declared as stable and, depending on its usefulness, a stable version may become unavailable once the appropriate period expires. One idea here is that some versions are *'more equal'* than others, and that if some critical bug happens to be found then the stable versions will be prioritised whilst the non-stable versions could even be pulled out, temporarily or otherwise. That is why the licensing terms state that packages relying on Qgraf should be able to use a stable version — not necessarily in an exclusive way. The interpretation of that part of the licensing terms should be that *some available version of the package should be able to rely on an available, stable version* of this program. If the package dependence is introduced[19] at a time when the latest version is not stable, and if no stable version fulfils the necessary requirements, then it could be acceptable to defer the fulfilment of that condition until the following stable version is declared — provided the inherent risks are assumed.

Non-stable versions should be expected to become unsupported, and unavailable for downloading, two years after the release of the subsequent stable version (but they might be pulled out for other reasons too).

There is now a provision for re-distributing officially *unavailable, stable* versions[20] in two cases (see the header of one of the latest Fortran files for details), provided there is no legal impediment: (*i*) with (old) packages for which no upgrade is feasible, or (*ii*) with either new or old packages, should Qgraf's official website close down. Nevertheless, in case (*i*) this provision might not be needed in practice as there is little backward-incompatibility in `qgraf-3`, specially at the level typically required by other packages.

[19] The relevant date is the release date of that package version, obviously.

[20] This means the latest patch version, of course; for example, `qgraf-3.4` is stable but, unless some other patch version is released in the meantime, `qgraf-3.4.2` would be the actual version to re-distribute.

41. Implementing the new specification

Input files accepted by `qgraf-4` are not entirely compatible with those accepted by `qgraf-3.6`, hence a conversion tool could be useful. The differences between the respective specifications have been described in Section 0. This section shows how to perform some of the necessary changes, in some cases using an integrated tool (in development). This tool is 'temporary' — that is, it should be deleted at some point. Currently, that tool can do (only) the following:

- add any required statement continuation characters to model-files and control-files;
- substitute the annotation marks by another such mark (a fixed one, that is), in any type of input file;

Other features will likely be added later. At least for now, the file to be processed is (roughly) assumed to be a valid file of the expected type (meaning that its contents are not properly checked); the tool might evolve, however. On the execution of the commands shown below, an empty line might be added either at the start or at the end of the new file (or both); trailing spaces will not be kept.

A `bash` script (ie file `q4in`) provided with the latest pack may help one to keep using current model-files and style-files with `qgraf-4.0` (Section 41.5).

41.1 Converting style-files

The conversion of a style-file can be performed by QGRAF itself, if run with a command of the following kind.

```
qgraf --s3to4 styles3/a1.sty styles4/a1.sty --percent
```

The first filename (here, `styles3/a1.sty`) should be the name of the style-file to convert, and the second the name of the new file. The options `--hash` and `--percent` define the annotation-mark to be used in the new style-file (respectively '`#`' and '`%`'); one of those options is required.

41.2 Converting model-files

Similarly, converting a model-file can be done with a command of the following type, where the conventions are analogous to those that apply to style-files.

```
qgraf --m3to4 models3/qcd models4/qcd -hash
```

41.3 Converting control-files

The conversion of control-files is also a partial one, at least at this stage. The respective command is similar to those to be used with the other types of input files, eg

```
qgraf --c3to4 qgraf.dat qgraf.in -percent
```

41.4 Addressing rules 4 and 5, easily

Some of the recently introduced backward-incompatibilities — ie those that involve deleted defaults of some kind — can be easily dealt with, even (or perhaps *specially*) in the case of automatic set-ups. Moreover, the simplest solution is (in each case) independent of the program's version; for thoroughness only, they are detailed below.

If the default external momenta are (implicitly) being relied on, rule 4 can be satisfied by simply making the corresponding (explicit) declarations in the `in` and `out` statements, eg

```
in = electron[p1], positron[p2] ;
out = photon[q1], photon[q2] ;
```

whether needed or not (and this last bit applies to the next two cases as well). Moreover, if the default integration momenta prefix is being used (implicitly), it is sufficient to always include the following statement.

```
loop_momentum = k ;
```

In auto-mode, if the default filename for the control-file is being relied upon, rule 5 can be satisfied by launching the program with a command-line argument equal to that default (possibly using a shell *alias,* if convenient), eg

```
qgraf qgraf.dat
```

This should work even for `qgraf-3.4`, in which case that argument will *not* be read. The point here is that there is a simple generic solution, which may simplify the update of related software; of course, it is also possible to detect the version of the program being used and address the various cases one by one. In API-mode, that filename has to be provided in the initialization call.

41.5 Additional remarks

Ideally (if time allows), when no recent features are needed, and specially in the case of automatic set-ups, it should be possible to run `qgraf-4` using input files compatible with older versions (say, `qgraf-3.6`). That would be achieved with at most three brief, additional, preliminary runs which would convert each 'oldish' input file into a compatible file — namely, by executing commands of the above described types. Then the diagram generation would take place in one further run, employing the input files produced on-the-fly. Naturally, any remaining constraints described in p. 4 (eg rules 1 and 5) would have to be implemented by the user.

The latest package includes a `bash` script file (ie `q4in`) that includes an implementation of the above mentioned ideas. Before running that script it is necessary to edit the file and adjust the definitions of the variables `OMFILE`, `OSFILE` and `QGRAF` to their environment. These variables should denote (respectively) a temporary model-file, a temporary style-file, and the executable for the program (path included). After that, the program may be launched (eg) as

```
./q4in models/qed styles/my_style my_control-file
```

assuming that the files defined by `OMFILE` and `OSFILE` are the ones also declared in the control-file; these two variables may remain fixed, of course, since the relevant ones are the arguments of the above command.

42. Obsolescent features

This section is likely to change in future versions of this guide, since it shall describe some changes that ought to become mandatory with `qgraf-5` (assuming that that version will exist at some point). The aim of such announcements is to try to preclude abrupt changes — by allowing a long period (years, possibly) for the implementation of the respective specifications, ahead of their enforcement. Changes that are still not fully determined will be included, provided they *are* planned. The current list follows.

(*1*) the diagram option `topol` will be deleted (`new_topology` can be used instead);

(*2*) the diagram option `simple` will be deleted (it can be replaced by a combination of two others);

(*3*) statements with an 'unbounded' number of arguments (Section 20) will be deleted; in their place there will be slightly more general statements depending on a single function, which will not have to be non-negative (even now any `iprop` statement can be replaced by a `psum` statement based on an appropriate p-function).

43. Changelog for qgraf-4

4.0.0 (May 2024)

New features: the statement continuation mark and long(er) input lines (p. 14); the config option `delete` (p. 21); the `output_dir`, `style_dir`, `model_dir` and `separator` statements (pp. 25 *f*); the options `new_elinks`, `new_loops`, `new_partition`, `new_topology`, `noedge` and its dual (p. 34); the style-keywords `<loops1>`, `<loops2>`, `<local_symmetry_number>`, `<nonlocal_symmetry_number>`, `<field_sub>` and its dual (p. 88 *f*); the scope of the prologue and epilogue sections has been extended (p. 88); the `-no_extra_chars` command-line option (p. 95); a preliminary version of the file conversion tool (p. 116).

Previously unannounced features: option `norbridge` and (duplicated) options `nosbridge`, `nobridge`, as well as their duals (pp. 34–35), all of which seem to have been already available with `qgraf-3.3`; the ability to use commutation *numbers* (instead of *signs*) in propagator statements (p. 63), which appears to have been introduced with `qgraf-3.6.0`.

Other changes: some backward-incompatibilities were introduced (p. 4, rules *1–3*), others announced (p. 4, rules *8–9*); the `options` and the `loop_momentum` statements are now optional;[21] the display-output may include the numbers of propagator-type and of vertex-type sectors of the input submodel, and the diagram count for each loop order (pp. 96 *f*); not for the first time, a few parameters were increased.

4.0.1 (May 2024)

Bug fix: a problem with function values.

An additional warning about Fortran module files was added to this guide (p. 9).

4.0.2 (June 2024)

New features: the `<index_offset>` style-keyword (p. 88).

Bug fix: re-activates option `topol`.

Other changes: Occasionally, a minor speed-up may be observed. The file conversion tool was improved a bit (p. 116). A tentative *interface specification* was added to the respective version of this guide (and was later replaced by the 'definitive' specification).

4.0.3 (June 2024)

Bug fix: a problem with `vsum` that may occur in the presence of duplicate vertices.

4.0.4 (July 2024)

New features: the ability to exclude a fixed external momentum term from some expressions for the diagram's momenta (p. 29); the `zero_momentum` statement (p. 38); the style-keywords `<momentum_loop>`, `<momentum_term>` and `<dual-momentum_term>` (on which the momentum-loop construct is based, p. 90), and also `<zero_momentum>` (p. 88).

Bug fixes: a problem with the internal momenta (which were not always computed), and additional problems with `<full_time>`, `<raw_time>` and `<index_offset>`.

[21] The change has essentially been reversed for the `loop_momentum` statement (with `qgraf-4.0.5`).

4.0.5 (September 2024)

New features: config options `flush` and `no_ntnls` (p. 23); the `count_to` statement (p. 41); the style-keywords `<loop_momentum>` and `<diagram_counter>` (p. 89); an implementation of the 'definitive' API (pp. 98 *ff*).

Bug fixes: some non-decomposable models were deemed decomposable; the `<command_data>` style-keyword would output (the) wrong statements in multi-output configurations; a problem involving the config option `noinfo`, which could cut the diagram generation short, apart from not blocking part of the display-output; reverts the redefinition of `<diagram_index>` in the epilogue.

Other changes: additional backward-incompatibilities were either introduced (p. 4, rules *4–5*) or announced (p. 4, rules *6–7*); the source code now includes directives, and some preprocessing options are necessary for compiling `qgraf-4` (p. 9); the rule for the implicit display-mode has been modified (p. 22); the file conversion tool has been improved a bit further (p. 116).