

QGRAF 3.6.10

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

Abstract

This document¹ is an addendum to the guide for **qgraf-3.0**, and describes what has changed since that version was released. To find out what is new in **qgraf-3.6** please see the Changelog (last section).

The file **qgraf-3.0.pdf** is still the primary documentation source. The extended discussion on the diagram sign computed by the program, previously available as a separate file, is now part of this document (Section 23).

¹ This version is part of the **qgraf-3.6.10 pack** (July 2024).

Contents

0.	Preliminaries	3
1.	The <code>config</code> statement	4
2.	The <code>messages</code> statement	7
3.	Multiple <code>output</code> (and <code>style</code>) statements	8
4.	The <code>loops</code> statement (extended form)	9
5.	The <code>index_offset</code> statement	10
6.	The <code>partition</code> statement	11
7.	The <code>psum</code> and <code>vsum</code> statements	12
8.	The <code>elink</code> statement	14
9.	The <code>plink</code> statement	16
10.	Option <code>bipart</code>	17
11.	Option <code>cycli</code>	18
12.	Options <code>nodiloop</code> , <code>noparallel</code> , <code>noselfloop</code>	20
13.	Option <code>onevi</code>	21
14.	Option <code>onshellx</code>	22
15.	The keywords <code><full_time></code> and <code><raw_time></code>	23
16.	The keywords <code><new_loops></code> , <code><new_partition></code> , <code><new_topology></code> , and <code><new_elinks></code>	24
17.	Duplicate vertices	25
18.	An extended language for describing models	26
19.	Command-line arguments	31
20.	Additional changes	32
21.	An example of a modern control-file	34
22.	An example of a modern model-file	35
23.	The diagram sign (extended version)	37
24.	Models with explicit propagator mixing	42
25.	Compiling	44
26.	Automatic downloads, licensing	46
27.	Changelog for <code>qgraf-3</code>	48

0. Preliminaries

The main feature introduced in **qgraf-3.6** consists in a set of new statements that often allow a more compact description of the input model. This development was intended for **qgraf-4.0** but a couple of related requests/suggestions ‘by’ Matchmakereft and FeynCalc accelerated its implementation. The ability to generate diagrams for more than one *cycle rank* (ie ‘number of loops’, real or apparent) in the same run is a relatively minor yet convenient feature which was probably long overdue. Other recently introduced features are mentioned in the Changelog.

In the last five years or so, the number of lines of Fortran code has grown steadily with each new (non-patch) version, the latest count exceeding 15 800 lines. By comparison, **qgraf-1.0** had just slightly over 3 300 lines, and even **qgraf-3.1.4** (the final version from the first development ‘era’), had less than 8 900 lines. Obviously, these numbers do not tell the full story — not every line of code survives from one version to the next.

Some of the developments began with suggestions, requests, or comments from a number of people. Concerning the more recent versions (ie **qgraf-3.1.5** and later), Lance Dixon suggested what became the diagram option **onshellx**, and Vladyslav Shtabovenko requested allowing more than one output-file, as well as allowing the control-file name to be specified as a command-line argument. Regarding running-time optimizations, John Gracey and V. Shtabovenko suggested improving the efficiency of (respectively) the graph generation for ‘large’ orders of perturbation theory, and diagram filters such as **vsum**. The first suggestion to extend the **loops** statement is likely due to Jos Vermaseren sometime in the nineties!

Although compiling QGRAF is (almost) as easy as before, there is now a short section about that (Section 25). Section 26 discusses automatic downloads.

In what follows, ‘*screen*’ (or ‘*terminal*’) means *standard output*, redefined or not, and ‘*to display*’ means *to send to the standard output* — computer monitor or otherwise. By default, the *control-file* is the file **qgraf.dat** in the working directory (although it is now possible to specify a different filename as a command-line argument).

1. 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.

1.1 The screen-modes

Given the current number of possible warning messages, it may be convenient to have some control over the information displayed. Three distinct *screen-modes*, defined by the keywords

```
noinfo
info
verbose
```

have been created for that purpose, and may be described as follows:

- **noinfo**: nothing should be displayed unless an error condition is detected;
- **info**: the program's name and version numbers, the input statements from the control-file, and the number of generated diagrams (per vertex-degree partition, and also the total number) are displayed in that order; 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 showing 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 can, as well as a summary of the model consisting of various numbers of propagators and vertices; as it should be clear, this mode may help fixing defective input files.

The screen-mode can be set by including one of those keywords in the `config` statement, eg

```
config = info ;
```

If the screen-mode is not set in the control-file then it is set implicitly:

- if an output-file is to be created, the screen-mode is set to **info**;
- else, the screen-mode is set to **verbose**.

If a run-time error is detected, the corresponding error message is displayed and then the program stops, irrespective of the screen-mode.

1.2 Improving the write performance

If the operating system is one of those for which a *newline* consists of a **line feed** (LF) control character (ASCII character 10), there is an experimental feature that should speed up the write operation and which is enabled by the keyword `lf`, eg

```
config = lf ;
```

This should yield a small to moderate (overall) performance increase, provided the output-

file is large enough. Compatible operating systems are listed in Wikipedia's *Newline*² page, Section *Representation*, and include Linux/GNU based systems. One may always run the program with and without that option (a single test case might be sufficient) and check if the output-files are identical.

1.3 Other configuration options

The keyword `noblanks` instructs the program to discard 'blanks' (ie each and every space character) in the filenames read from the control-file. This means (eg) that the statement

```
output = ' ' ;
```

becomes equivalent to

```
output = ;
```

irrespective of the operating system.

The keyword `nolist` disables creating the output-file(s) even if one or more filenames are declared, eg

```
config = info, noblanks, nolist ;
output = 'dlist' ;
style = 'f0.sty' ;
...
```

1.4 A missing option

When the number of diagrams is 'big' (larger than 10^5 , say), the output-file will also be quite large, obviously, and it might be desirable to split it into 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 may not be a real problem as there exist tools which, with little effort, can be used for the job. The remaining of this section describes a possible method (for Linux/GNU operating systems).

The following lines

```
#_<diagram_index>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 which 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 the line command

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

This splits output-file `dlist` into smaller files (the *pieces*), containing the description of 10^4

² <https://en.wikipedia.org/w/index.php?title=Newline&oldid=863813417>

diagrams each (except for the last piece, which could have a smaller number), as the pattern 0000xyx appears every time the diagram index is a multiple of 10000. 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 arguments of that command. To keep the number of files within reasonable bounds, the number of diagrams per piece should typically be larger than 1% of the total number of diagrams (hence having 10^4 diagrams per piece might be acceptable if the total number of diagrams does not exceed one million, say).

That process may 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 into 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.

If the extra lines are processed as commentary by the program that reads them, nothing else needs to be done; if not, they can be eliminated with the help of a simple **bash** script, eg

```
TMPF="xx0tmp"
\rm -f $TMPF
for xf in `ls xx*` ; do
    \grep -v xyx $xf > $TMPF
    \mv -f $TMPF $xf
done
```

It is safer to do the whole procedure in a directory containing only the necessary files, of course. The prefix should be chosen with care, as **csplit** overwrites existing files.

In the future, hopefully, a direct interface to the diagram amplitudes generated by QGRAF should provide a reasonable solution to the problem of storing and/or processing the program's output in more diverse ways. Meanwhile, one may sometimes use the **partition** statement to lessen the current difficulties.

2. The messages statement

The `messages` statement declares the name of a file to be created by the program for saving warning and error messages. This statement is optional, independent of the screen-modes, and aimed mainly at script based setups. It may appear only once, either immediately below the `config` statement, eg

```
config = info ;  
messages = 'msg.txt' ;  
output = 'dlist' ;  
style = 'f0.sty' ;  
...
```

or, if the `config` statement is absent, as the first statement in the control-file.

Note, though, that messages issued before the messages-file has been opened will not be stored in that file. Problems at that stage should occur only rarely, if at all, unless (i) the control-file cannot be read or contains errors (ie the `config` or the `messages` statements contain errors), or (ii) the messages-file already exists or cannot be opened. The file will not be saved at the end of the run if it contains no messages; moreover, the symbol (**w!**) mentioned in Section 1.1 will not be displayed if such a file has been created.

3. Multiple output (and style) statements

To generate multiple (up to four) output-files in the same run, style-files and output-files should be paired (their names appearing *alternately*) eg

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

All of those files should be distinct. A valid style-file name is required for each pair, eg a 'trivial pair' such as

```
output = '' ;
style = ;
```

is not allowed (but it is possible to have *zero* pairs).

NB: for each output-file, the <command_loop> skips those iterations corresponding to **output** and **style** statements that do not refer to that file.

4. The loops statement (extended form)

From the very start, the `loops` statement required users to specify exactly one value (ie the *number of loops*, or *cycle-rank*) for the diagrams to be generated, 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 diagram 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 , where $d \geq 3$, contributes with weight $(d/2)-1$ (which is positive). For example, at tree-level, in a simple gauge theory, cubic vertices are proportional to the coupling constant g whilst quartic vertices are proportional to g^2 (and propagators do not depend on g). Actually, in what follows, it may be more convenient to consider the expansion of the amplitude in g (or in some other appropriate factor) rather than on the number of loops, so that the weights become integer (in which case they can be accepted by `vsum` and `psum` statements).

Nonetheless, there are cases for which the number of loops given as input does not represent the order of perturbation theory — because the vertices and/or propagators may represent some higher order corrections instead of the tree-level interactions and propagators — and which require dealing 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.

However, there is a marked difference between dealing with usual and unusual weights: 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* a single (explicit) weight, each run provides the required diagrams for just one such order.

NB: The output sent to the screen 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.

5. 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 (optional) `index_offset` statement instructs the program to add a non-negative integer to the default diagram index, eg

```
...
options = ;
index_offset = 1071 ;
```

This offset is disabled in the epilogue section, where the (style) keyword `<diagram_index>` is still replaced by the number of diagrams listed in the output-file. This statement may appear after the `options` statement, but before any `true` or `false` statement.

The following example shows a possible application of the `index_offset` statement. Let us suppose 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 of two or more conditions, eg

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

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

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

then (case 2)

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

and finally (case 3)

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

which may be run separately. Similarly, the exclusive disjunction can be divided into two non-overlapping cases, the equivalence $A \Leftrightarrow B$ into two cases also, and so on.

6. The partition statement

Unless the `noinfo` option is enabled, the output displayed includes the (*vertex*) *degree partitions* that are simultaneously compatible with all of the following inputs: the model, the physical 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 *topology* with that 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 subsets of those partitions. It may appear at most once, after the `options` statement and before any `true` or `false` statement. For example, if a model has cubic and quartic vertices only, the statement

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

requires the diagrams to have precisely two vertices of degree 3 and one vertex of degree 4. Let us now suppose that the set of vertex degrees of the input model is $\{3, 4, 5, 6\}$. Then, since any missing term is assumed to be indeterminate, 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. To pick a single partition it may be necessary to specify nearly all (or even all) of the vertex degrees and their multiplicities. A ‘term’ n^k should not appear in the `partition` statement unless the input model contains some interaction vertex of degree n . In the above examples each term sets the exact multiplicity k of some degree n , but it is also possible to impose inequalities instead of equalities. For example, the statement

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

demands at least one cubic vertex, exactly one quartic vertex, and at most one quintic vertex. On the other hand, the constraint defined by the statement

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

is trivial.

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 example matches the first of the above statements.

```

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

total = 10 connected diagrams
```

7. The psum and vsum statements

The ability to define functions (ie parameters associated to fields, propagators, and vertices) in the model-file implicitly suggests new diagram selection criteria. It is now possible to impose some numerical constraints that depend on integer functions. 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; in fact, partial radiative corrections based on subsets of diagrams defined by such conditions have been routinely considered in the particle physics literature. In practice the new filters consist of optional statements that may appear in the control-file (see file `qgraf-3.0.pdf`, Section 4.3).

7.1 The vsum statement

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 10^4 , say).

Obviously, this filter may 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 `binary` a v-function that maps vertices in V_1 to '1' and vertices not in V_1 to '0'. Then, the statement

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

selects diagrams containing at least one vertex in V_1 .

7.2 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.

There is a nontrivial difference between the `psum` and `vsum` statements, apart from the fact that they operate on different types of functions. As QGRAF does not produce tree-level propagators any generated diagram has at least one vertex; therefore any implicit summation

for `vsum` includes at least one term. However, there exist tree-level diagrams without propagators — namely, the diagrams representing the tree-level vertices of the model, which we may call *stars*. There are also tree-level processes with diagrams of ‘mixed-type’, ie some of the diagrams (non-stars) have one or more propagators whilst the remaining diagrams (the stars) have none.

Conventionally, QGRAF assigns a null term 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).

7.3 Additional comments

Asymptotically, ie for a large number of vertices (`vsum`) or propagators (`psum`), these filters do not (moreover,³ *cannot*) have efficient implementations, if they are general enough. For some particular cases, though, more efficient algorithms do exist — nonetheless, they have not been implemented yet (there has been an improvement with `qgraf-3.5`, but a relatively small one).

Update: `qgraf-3.6` brings additional efficiency gains for both `vsum` and `psum`; apart from that, in some cases this efficiency issue can be surmounted by defining *submodels*.

³ At least assuming the widely held belief that the algorithmic complexity classes P and NP do not coincide (ie $P \neq NP$).

8. The `elink` statement

The `elink` statement provides a way to restrict the configuration of external lines — ie 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, may be simulated by constructing an extended model — defining new **external** fields and new vertices — but that approach can be exacting.

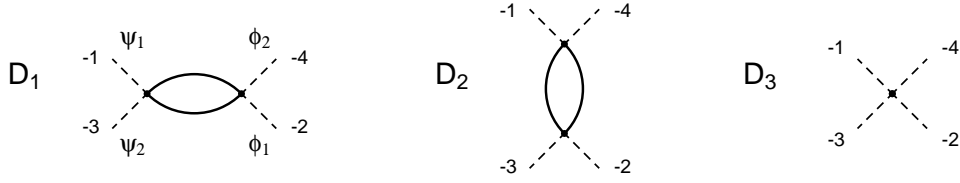


Fig. 1. Illustrating the `elink` statement.

Let us take a look at some examples, assuming the generic 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 to 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 (Fig. 1), 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 (in total) to 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 should link to *other* vertices. This condition rejects 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. If k denotes the number of arguments of the first type, and n the number of legs, then the inequalities

$$1 \leq a \leq b \leq k \leq n, \quad n \geq 2,$$

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

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

requires ψ_1 to link up with some other external field(s), whilst

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

requires ψ_1 not to link up with ϕ_1 (a condition which diagram D_3 fails to satisfy). 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.

9. 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 those 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*’ (like a selective `onshell` option, see below).

If a connected diagram has a regular bridge then deleting that edge will split the set of external fields into two non-empty subsets X_1 and X_2 , each one containing the external fields belonging to one of the two sub-diagrams obtained. If \mathbf{p}_i and \mathbf{q}_j denote the incoming and outgoing momenta, respectively, then the momentum \mathbf{P} flowing through that edge can be written as

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

with $a_i, b_j \in \{0, 1\}$; here, 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 $\sum \mathbf{p}_i = \sum \mathbf{q}_j$, it should be clear that \mathbf{P} can be expressed as a linear combination of the external momenta in two different ways.

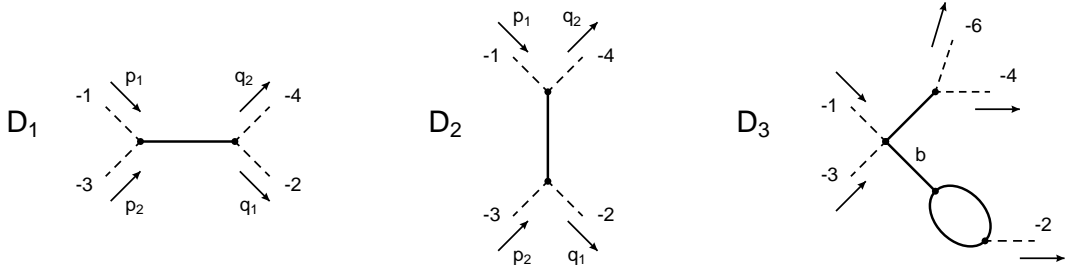


Fig. 3. Illustrating the `plink` statement.

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 $\mathbf{p}_1 + \mathbf{p}_2$, like diagram D_1 (Fig. 3); 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 may also be used to set only part of the external fields ‘*on-shell*’. For instance,

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

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

10. Option `bipart`

The option `bipart` selects those diagrams whose topology is a bipartite graph — ie a graph G 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 requiring that G has no circuit (or *loop*) of odd length; in particular, G must have no self-loops. Every tree is bipartite.

The dual option is `nonbipart`.

11. Option `cycli`

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. 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_2 and D_3 (Fig. 4). 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 , 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 .

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 $k_1, k_2 \dots k_l$ be the independent integration momenta, and ignore the external momenta (eg set them to zero). Then each cycle-block is a minimal,⁴ 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, are independent of each other (ie 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. For example, any self-loop defines a cycle-block.

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; the dual option is `cyclr`. 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. Fig. 4 shows a few examples. Diagram D_1 has two self-loops and thus two cycle-blocks (hence it will be 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.

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 will be rejected too. That type of partitioning is not possible for diagram D_3 , which will be validated; D_4 will also be validated

⁴ ie non-decomposable into two or more sets of the same kind

by `cycli`, just like any tree or 1-loop diagram.

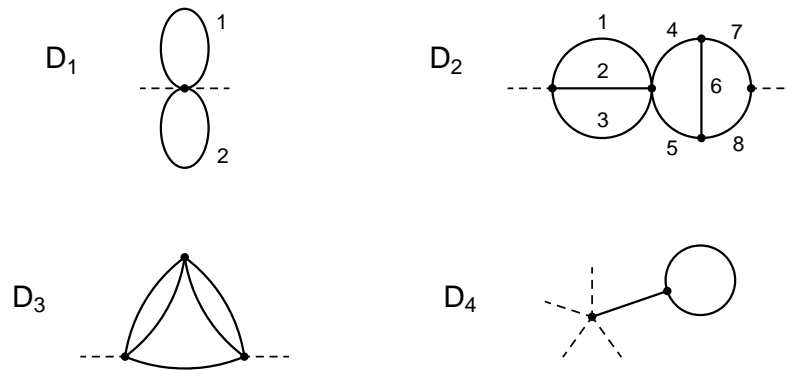


Fig. 4. Illustrating option `cycli`.

12. Options `nodiloop`, `noparallel`, `noselfloop`

There are three pairs of options to deal with circuits of length one (*self-loops*) and two (*di-loops*):

- `noselfloop` excludes (diagrams with) self-loops.
- `nodiloop` excludes di-loops (ie any two distinct vertices may be joined by at most one propagator).
- `noparallel` excludes parallel propagators (ie no di-loops, and each vertex can have at most one self-loop).

Their duals are (respectively) `selfloop`, `diloop`, and `parallel`. The equivalence

$$\text{simple} \leftrightarrow \text{noselfloop}, \text{nodiloop}$$

should be clear.

13. Option `onevi`

Option `onevi` 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 (interaction vertex, that is). 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; moreover, if both bridges and self-loops are excluded then they become identical.

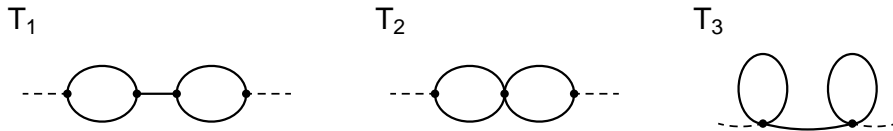


Fig. 5. Illustrating option `onevi`.

In Fig. 5, T_1 and T_2 are 1-vertex reducible: deleting any vertex that does not connect to an external line will generate two disjoint components. Topology T_3 is 1-vertex irreducible as it has too few vertices to be reducible. The previous figures provide useful examples too: the 1-vertex reducible topologies are T_3 in Fig. 3, and T_2 in Fig. 4.

There is a class of topologies that are both 1-particle reducible and 1-vertex irreducible, for which T_3 (Fig. 5) 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 lines connect to u and/or v , obviously.

A minor technical point: usually (as per classical graph theory) the deletion of a vertex implies also the deletion of the edges incident to that vertex. A slightly different option, which seems more natural in a QFT context, consists in deleting a vertex and all its incident half-edges, after having first cut each incident edge into two half-edges. Obviously, this does not change the above definition of 1-vertex (ir)reducibility.

14. Option onshellx

The 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 the momentum on one of the external lines (as in diagrams D_2 and D_6 (Fig. 6), 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. The 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 fused (compare eg D_1 and D_2). Diagrams rejected by `onshellx`, but not by `onshell`, will have at least one vertex of degree $d \geq 4$. Like most of the other options, `onshellx` has a ‘topological’ character: only the diagram topology matters, not the actual fields.

Diagram D_1 will be 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 will be discarded by either option, obviously.

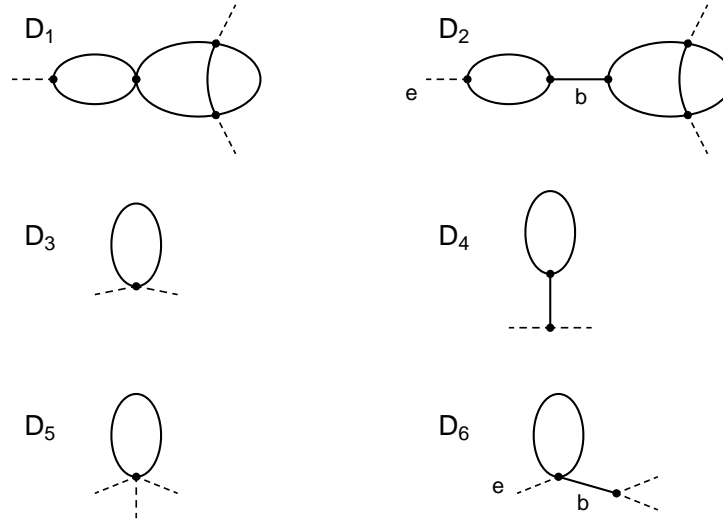


Fig. 6. Illustrating option `onshellx`.

Diagram D_3 will not be rejected by `onshellx`, unlike D_5 : although D_3 may 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.

The diagrams selected by `onshell` and rejected by `onshellx` follow from the combination

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

and this allows one to do a basic cross-check in the case where those diagrams are thought to evaluate to zero. 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.

15. The keywords <full_time> and <raw_time>

The keyword <full_time> may appear in the prologue and epilogue sections of the style-file. The output generated by this keyword 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:

```
2022/09/16 10:15:15.389 +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

```
20220916 101515.389 +0100
```

The obvious usefulness of these keywords consists in being able 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). When there are no issues, inserting any of those keywords in both of the above mentioned sections may give us a very good estimate of the *real time* taken by the execution of the program (namely, by computing the elapsed time from the respective outputs). That elapsed time might not include the (total) amount of time taken by the operating system in saving the output-file in the computer disk, as that file is usually created in the computer RAM and often not immediately copied to the disk.

16. The keywords `<new_loops>`, `<new_partition>`, `<new_topology>`, and `<new_elinks>`

There are three new keywords for the diagram section of the style-file. They are not ‘normal’ data keywords, though, as they do not depend only on the current diagram.

The output generated by `<new_topology>` specifies whether or not the topology of the current diagram differs from the topology of the preceding (output) diagram; a 1 is produced when they differ, else a 0 is produced (and for the first diagram, a 1 is produced). Note that the ‘topology’ of a diagram is independent of the arrangement of the external fields, but depends on the number and arrangement of the external lines (ie with fields omitted).

The keyword `<new_elinks>` produces a 1 in the following cases (and a 0 otherwise):

- when the keyword `<new_topology>` produces (or would produce, if present) a 1,
- when the keyword `<new_topology>` produces (or would produce) a 0, *and* 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, let us point out that `<new_topology>` might be employed to speed up the diagram topology identification. On the other hand, if the model has a single field (and if that field is neutral) 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 divided among multiple output-files, or if those diagrams were sorted in a different order.

In the same vein, the keyword `<new_partition>` produces a 1 not only for the first diagram, but also for any subsequent diagram for which the vertex degree partition differs from that of the preceding diagram (otherwise, it produces a 0).

Finally, the keyword `<new_loops>` may be useful when the `loops` statement requests diagrams for more than one cycle-rank (ie *number of loops*). This keyword produces a 1 for each diagram whose cycle-rank differs from the cycle-rank of the previous output diagram (and also for the very first diagram), else it produces a 0.

17. Duplicate vertices

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

Duplicate vertices have been tolerated for some time but typically QGRAF generated more diagrams than necessary (with the appropriate ‘symmetry factors’, nevertheless). This has been improved with **qgraf-3.3** — further symmetries are taken into account in this case, and the number of diagrams may now be smaller.

18. An extended language for describing models

The input-model description has been overhauled, and now one may define *sectors*, *submodels*, and *default values* (or simply *defaults*) for functions. What may be achieved with the new statements includes (i) a more compact description of ‘complex’ models (ie models with many propagators and/or vertices, and for which one or more functions have been defined), and (ii) a combined description of certain models and their extensions in a single file. Creating a model-file with submodels 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 submodels by editing a single file.

Model-files accepted by (eg) **qgraf-3.4** continue to be valid, but if a model file includes any of the new statements described in this section then the new specification applies *in toto* (which implies that constants and functions will have to be declared explicitly).

A model-file is divided into six *zones* (ie implicit sections), but in some cases some of them may be empty. An illustrative ‘toy’ example of a modern model-file — that may be useful in figuring out the sequence in which the different types of statements should appear — can be found in Section 22.

18.1 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, and each type contains at most two (other) types of statements. A *propagator-type* sector contains 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, + ; pf2= 'A' ]
end[ sector1 ]
```

In a similar fashion, a *vertex-type* sector may contain statements that define a default value for some v-function, and must contain one or more vertex statements, eg

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

Propagator and vertex statements need not be part of some sector block. Nevertheless, 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 must be a single statement for each type of sector, eg

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

18.2 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 should be declared at the very top of the model-file (zone 1), before being defined one by one in zone 4. For example, the statement

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

declares two submodels, and the statement block

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

defines submodel `subm1` by listing the sectors that it comprises (their relative ordering in this statement being immaterial). A submodel may 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 may be defined either by listing all the sectors in an `include` statement or by using a ‘void’ `exclude` statement, ie

```
[ exclude :: ]
```

Such a submodel has to be defined if the complete model-file describes a (sub-)model that QGRAF should be able to use and if at least one other submodel is defined. Concerning the control-file, one may either have the usual type of `model` statement if no submodel is defined in the model-file, eg

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

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

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

18.3 Constants

For a model-file that does not define any submodel the constants are (well ...) really *constant*. When the model-file defines two or more submodels it may be convenient to have *submodel-dependent* ‘constants’ (for example, one may wish to define for each submodel a

distinct string representing the name of that submodel). The constants should be declared in a single statement, in zone 2. The submodel-independent (or *global*) constants should be defined in zone 2 as well, after (ie below) the declaration statement. This is illustrated in the following example (note that only the first type of statement is new).

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

The submodel-dependent constants should be defined in every submodel block, in zone 4, unless defaults are defined (this alternative method is described in section 18.5). The syntax is as to be expected, for example,

```
submodel[ subm1 ]
  [ include :: sector1, sector3 ]
  [ c1= 'qed3' ]
end[ subm1 ]

submodel[ subm2 ]
  [ exclude :: sector5 ]
  [ c1= 'qed1' ]
end[ subm2 ]
```

The `include` (or `exclude`) statement should be the first statement in a submodel block.

18.4 Functions

Functions come in three types, and there is a distinct statement for declaring each type (in zone 3). 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. The `integer` qualifier should be added⁵ when declaring functions that may be involved in some integer constraint (implemented in terms of `vsum` and/or `psum` statements), eg

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

There can be at most *two* statements of this kind for each function type — one statement for the generic functions (to be treated as strings of characters) and one other statement for the integer functions, with no repeated identifiers between them.

18.5 Default values

If a function takes the same value many times (in a given model), it may be useful to define one or more default value(s) for that function. Default values may be defined globally

⁵ This is not being enforced straight away, but it should be enforced in future versions.

and/or per sector with the sectorial values taking precedence over the global ones. For each function, one may define at most *one* global default, and at most *one* default per sector. Sectors need not be defined for this purpose if all defaults are global. Still, if *any* default value is to be defined then *every* function *and* constant must be declared.

To define a global default for vertex function **vf2** (say), one simply has to state eg

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

in zone 3, after declaring that function. As a result **vf2** no longer has to be defined for every vertex — only those values that differ from the default (in this case, 1) have to be declared explicitly, in the usual way. Additionally, if the model-file contains sectors then sectorial defaults may be defined too, whether or not global defaults exist, eg

```
sector[ sector3 ]
[ ; vf2 = 0 ]
...
```

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. Defaults for distinct functions (and constants) should be defined in different statements. Having statements with either a single or a double semicolon may serve as a reminder that the corresponding default values can be overridden at one or two other stages, respectively.

Defining defaults for propagator-functions is similar, but for field-functions there are more possibilities. For instance, 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 (say) not every model (or sector) has both types of fields.

The precedence rules within each sector are as follows: statements defining f-function or p-function defaults should precede propagator statements; 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) must 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 one states

```
[ ; c1= 'qcd' ]
```

in zone 2, after declaring **c1** as a constant, then 'qcd' becomes the global default for **c1**. That default may be overridden in any submodel block, eg

```
submodel[ subm1 ]
[ include :: sector1, sector3 ]
[ c1= 'qcdx' ]
end[ subm1 ]
```

18.6 Additional remarks

Defining submodels requires introducing sectors as well, obviously, but not conversely. 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 every sector is *active*). The active sectors are those that the program will focus on, of course, and they are fully checked for syntax and content, whilst 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 in this extended language, one should try to steer clear of that path as much as possible. The proactive measures include adding an adequate amount of commentary, avoiding cryptic identifiers, defining the sectors carefully, and grouping together those propagator (or vertex) statements that are not part of any sector.

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 happen in late 2024).

19. Command-line arguments

For `qgraf-3.5.2` and later versions the command-line interface can be used to pass a few basic instructions to the program. Executing either of the following commands

```
qgraf -version
qgraf --version
```

should result in QGRAF displaying its own name and version numbers (typically left-aligned on the standard output and terminated by a *newline*), and then exiting at once. To give an example, for `qgraf-3.6.10` the output string should be

```
qgraf-3.6.10
```

For a normal run, it is possible to define the control-file name as a command-line argument, eg

```
qgraf my_control-file.in
```

in which case the program will read the file `my_control-file.in` instead of the standard one (`qgraf.dat`). Still, there are some restrictions⁶ on the filenames input in this way:

- 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* (all of which appear in the above example);
- the first and the last characters must be alphanumeric;
- no two special characters should appear consecutively.

⁶ Some initial implementations may impose weaker restrictions, but these are the ‘official’ ones.

20. Additional changes

In addition to the features discussed in the previous sections, there are other differences between **qgraf-3.6** and **qgraf-3.0**, and the most relevant of those are described below.

There is a different presentation of the model partitions, which now includes some additional types of propagators (this refers to the output displayed). Here is an example

```
propagators: (8)    2N+  3C+  1N-  2C-
```

which is to be interpreted according to the following rules:

- The signs ‘+’ and ‘-’ denote the propagator sign (ie, whether the propagators are defined in terms of commuting or anti-commuting fields).
- 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.

For instance, the propagators of Dirac fermions (and the propagators of some ghost fields as well) contribute to the coefficient of **C-**, while the propagators of Majorana fermions contribute to the coefficient of **N-**. Therefore, in the above example the model-file defines eight propagators, five ‘bosonic’ and three ‘fermionic’; two of those bosonic propagators are neutral (string **2N+**) and the other three are charged (string **3C+**); 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 declaration contains 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)    1N+
propagators: (7)    1N+  3C+  1N-  2C-
```

Those numbers are mutually exclusive (in this example the model-file contains eight propagator declarations too).

For **qgraf-3.2** and later versions, and for 1-point diagrams only, the behaviour of options **nosnail** and **snail** is not the same as for earlier versions: instead of rejecting all such diagrams, **nosnail** is in that case equivalent to **onshellx**.

The implicit definition of option **floop** has not remained completely fixed since its original implementation. 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.

The diagrams rejected by **floop** are those that have at least one loop (ie circuit) of odd length for which every propagator is that of an anti-commuting field. The idea behind **floop** is Furry’s theorem for QED, but it is up to the user to decide on its use. The dual option **notfloop** is also available.

Input files containing a so-called **UTF-8** byte order mark, which is included by some applications when saving a text document, have been accepted since the release of **qgraf-3.1.4**. This provides the ability to create input files with a broader range of software (the actual text characters must still be the printable ASCII characters, obviously).

20.1 Obviating the diagram generation

In some cases the program is able to detect that there are no diagrams, either for certain vertex-degree partitions or even for all such partitions, without trying to generate any diagram (this is work in progress). Those situations can be created by some combinations of options and/or statements. In such cases (those detected, that is) the output displayed contains additional symbols. If the program detects a complete absence of diagrams (for instance, if the diagrams are required to be trees and also have self-loops), it will simply display

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

without listing any partial result; else, for each ‘flagged’ degree partition (ie a partition for which it has determined there can be no diagrams), the corresponding line in the output displayed will look like this

```
3^2 4^1      ....      0 **
```

In either case, the presence of those asterisks should mean that the usual diagram generation was not attempted.

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

20.2 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, there are also upper limits); values of the form $2^n - 1$ should be preferred.

21. An example of a modern control-file

```
%%  An example of a modern control-file
%%  with optional statements commented out

%  config = info, lf, noblanks ;
%  messages = 'msg.txt' ;
    output = 'qlist' ;
    style = 'f1.sty' ;
%  output = ;
%  style = 'f2.sty' ;
    model = qed1 // 'qedx' ;
    in = mu_minus[p1], mu_plus[p2] ;
    out = photon[q1] ;
    loops = 4 ;
    loop_momentum = k ;
    options = onepi, cycli ;
%  index_offset = 257 ;
%  partition = 3^5 4^2 ;

%%  some other constraints may follow, eg

%  true = vsum[ v_weight, 2, 4 ];
%  false = psum[ p_weight, 0, 1 ];
%  true = elink[ -1, -3, incl, 1, 1 ];
%  false = plink[ -2 ];
```

22. An example of a modern model-file

```

%      zone 1
%
% the submodels, and 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 submodel-dependent 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_functions :: vf1 ]

%      zone 4
%
% the submodel blocks

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

```

(continues into the next page)

```

submodel[subm2]
    [ exclude :: ]
    [ c2 = 'full model' ]
end[subm2]

%      zone 5
%
% the propagators and the propagator-type sector blocks

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

sector[sk1]
    [ ; pf2= 'B' ]
    [ psi, psibar, - ]
    [ lambda, lambda, - ; ff1= (1) ]
end[sk1]

%      zone 6
%
% the vertices and the vertex-type sector blocks

[ 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]

```

23. The diagram sign (extended version)

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. 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). QGRAF 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. 7. 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, spacetime indices, and so on) will appear in the Feynman rules.

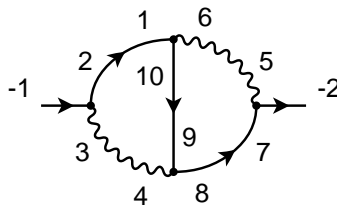


Fig. 7. A diagram (in QED) and its field index numbering.

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 — we will only perform a sequence of simple transpositions, each one involving exactly two fields. If we discard 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 of fields. The next point to consider is that, whatever the type of field, the propagator ordering chosen by QGRAF 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, viz

$$\Phi_{-2}^{out} \Phi_{-4}^{out} \dots \Phi_{-2s}^{out} \Phi_{-2r+1}^{in} \dots \Phi_{-3}^{in} \Phi_{-1}^{in},$$

where r and s denote the number of incoming and outgoing fields, and where the subscripts are (again) the field indices. Any occurring sequence S_e of external fields obtained in the previous step is then compared with the former one, and the parity of the number of transpositions t_e needed to convert S_e into the reference sequence (divested of its commuting fields) is determined. Hence a new factor $(-1)^{t_e}$ is generated, and the diagram sign will be equal to $(-1)^{t_p+t_e}$.

In our example, once the internal fields have been dropped from F_3 we are left with the sequence

$$F_4 = \Psi_{-1} \bar{\Psi}_{-2},$$

which may be converted into the reference sequence using a single transposition. Therefore, as $(-1)^{t_p} = (-1)^{t_e} = -1$, QGRAF generates the sign $+$ for that diagram.

In comparison with a graphical rule, the preceding definition of diagram sign might seem (to a human) unnecessarily complex, even error-prone; nevertheless, it is closely related to Wick’s theorem, and its computation presents no special difficulty. On the other hand, users who habitually rely on graphical rules might find some of the consequences of that definition rather unexpected. For that reason, we will now take a look at a number of issues that may arise out of misunderstanding or by mere inattention.

23.1 Problem 1

Let us suppose we have defined QED as follows,

```
% propagators
[ photon, photon, + ]
[ positron, electron, - ]

% vertex
[ positron, electron, photon ]
```

where `electron` corresponds to a Dirac field Ψ and `positron` corresponds to $\bar{\Psi}$. The usual declaration for that fermionic propagator is

```
[ electron, positron, - ]
```

as this is the one that corresponds to the ordered contraction $\langle \Psi \bar{\Psi} \rangle$. The reason why the former declaration may 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 QGRAF's output is replaced by the conventional propagator expression, the result may be an incorrect amplitude (depending on the exact substitution used). 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, owing to the following: first, for a fixed process, and a fixed order of perturbation theory, every diagram has the same number of fermionic propagators; second, when going from loop order n to order $n+1$, the number of fermionic propagators increases by *two*. Nonetheless, there exist many models for which one can find (for the same scattering process) two diagrams that have the same sign if the fermionic propagators are declared in one way, and opposite sign in the other way.

23.2 Problem 2

We want to compute the amplitude for some scattering process that involves fermions (in the initial or in the 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. A somewhat different problem may occur as well if the external momenta are not declared, of course.

23.3 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 incorporates the diagram signs computed by QGRAF, and decide to 're-orient' some fermion propagators directly in that program's output. For example, we do some substitutions such as

```
prop( Psi(3,k1), Psi(6,-k1) ) → prop( Psi(6,-k1), Psi(3,k1) )
```

where each of these expressions denotes a propagator function for a Majorana fermion `Psi`.

That may also be a problem 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 may also decide to exchange some arguments describing that same fermion, this time in some expressions that represent vertices. For instance, we have an amplitude which includes a sub-expression such as

```
vertex( Psi(5,k1), Psi(2,-k1-k2), A(7,k2) )
```

where A denotes some bosonic field, and then (simply) exchange the arguments $\text{Psi}(5,k1)$ and $\text{Psi}(2,-k1-k2)$ in that expression — since the fields are identical there is no problem, right? Wrong, that exchange would have affected the computation of $(-1)^{t_p}$.

23.4 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 may 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 process like

```
in = electron, positron ;
out = muon_minus, muon_plus ;
```

may sometimes be indistinguishable from (eg)

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

Here, QGRAF seems to compute the correct relative sign between the (former) ‘non-radiative’ process and the (latter) ‘radiative’ process, *if* the respective declarations match as above (one should check all the same, of course). Nevertheless, in general, QGRAF does not address that type of problem, ie other means must be employed to determine an appropriate global sign for each process. The sign of an amplitude may often be adjusted by permuting two incoming (or two outgoing) fermions, but that is a minor point.

23.5 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 otherwise problematic substitutions (instead of discarding QGRAF's sign altogether). The basic idea is that each substitution that can affect the diagram sign should also generate a factor (or an appropriate power) of -1 . In other words, any substitution involving an odd number of fermionic transpositions should generate a compensating factor of -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 completely, 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 is (very likely) 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 may help, but the diagrams are then listed in the same output-file (hence they have to be selected afterwards depending on the order of perturbation theory being computed).

Solutions 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 this also applies to the other issues).

23.6 Further comments

Graphical rules were first created at a time when Feynman diagrams were generated by hand, and they allowed the person doing the QFT calculation to easily (ie 'visually') determine a practical diagram sign. However, 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 setups being the norm, the need for graphical rules is much lower. 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 hide. Lastly, graphical rules do not necessarily solve Problem 4 either.

24. 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 declarations. 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

$$\mathcal{L} = \frac{1}{2} \partial_\mu \phi_1 \partial^\mu \phi_1 + \frac{1}{2} \partial_\mu \phi_2 \partial^\mu \phi_2 + a \partial_\mu \phi_1 \partial^\mu \phi_2 + \dots$$

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:

$$\phi_3 \rightarrow \phi_1, \quad \phi_4 \rightarrow \phi_2.$$

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). However, as QGRAF requires one to identify the incoming and the outgoing fields, 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

$$\phi_3 \rightarrow \phi_2, \quad \phi_4 \rightarrow \phi_1.$$

For example, since ϕ_3 and ϕ_4 constitute a conjugate pair, the input process

```
in = phi1 ;
out = phi3 ;
```

is essentially equivalent to

```
in = phi1, phi4 ;
out = ;
```

which (by the former substitution rules) is equivalent to

```
in = phi1, phi2 ;
out = ;
```

which (as ϕ_2 is self-conjugate) is equivalent to

```
in = phi1 ;
out = phi2 ;
```

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 elements. On the other hand, the process

```
in = phi1 ;
out = phi4 ;
```

is not equivalent to the previous ones.

24.1 The keyword `internal`

Although the keyword `internal` may seem not to be really needed it can be useful in describing some models, should one wish to specify that some fields should never appear as external fields of the physical process (or correlation function) given as input. For example, if the model-file contains the propagator statement

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

then `phi1` and `phi2` should both be excluded from any `in` and `out` statements, otherwise an error condition will occur. The keywords `internal` and `external` are incompatible, of course.

25. Compiling

The source file(s) of **qgraf-3.5** and later versions are expected to be compatible with the Fortran 2008 standard, although testing has been restricted to the GFortran compiler. 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 may place those files in a user-definable directory, and they can be deleted once the executable has been created. In a Linux/GNU system, one may execute (eg) the following line commands

```
mkdir fmodules
gfortran -o qgraf -O3 -J fmodules qgraf-3.6.10.f08
```

to obtain a binary named **qgraf** from a Fortran 2008 version (the correct version numbers should be used, of course). The option **-O3** 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; the option **-O2** optimizes a bit more (for performance) than **-O3**, and **-O3** is also a possibility (option **-Ofast** should *not* be used). If the compilation process generates an error like this one

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

that likely means that the compiler found earlier module files in a different format (which it could have ignored) and then it gets a bit confused. In that case, find and delete all the problematic module files, *both* in the working directory and in the directory specified by the option **-J** (as reported by the compiler, their filenames should have the extension **.mod**). Better yet, remove the Fortran module files before each new compilation takes place (there should be a dozen such files or so).

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... one should always check for pre-existing Fortran module files, and delete them before compiling the source code — or else create a new directory (and a subdirectory) for that purpose, as already mentioned.

As there are no modules in Fortran 77, the next command may be used as a template for compiling older versions.

```
gfortran -o qgraf -O3 qgraf-3.4.2.f
```

There are executable/binary versions of GNU Fortran 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).

The license for this program has changed ‘recently’ (see Fortran file). Should you find something in it that you really consider unreasonable, please let me know.

Compiling with options that rely on somewhat sophisticated code transformations to produce more efficient binaries (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 (with similar but possibly computationally less-demanding cases), where the outputs of two or more binaries compiled with different optimization levels are compared — a binary with the desired ‘aggressive’ level, and one or more binaries with ‘non-aggressive’ levels, say. Furthermore, it should be kept in mind that changing the program’s environment (hardware or software, eg just updating the compiler, which one’s computer might do automatically, and then recompiling) may invalidate a previous test.

It is perhaps worth mentioning that QGRAF does not make system calls to run shell commands, ie it does not rely on statements of the following forms.

```
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 2008. Obviously, input and output statements are used extensively, providing access to the standard output, to the control-file, and to any file specified in the control-file.

26. Automatic downloads, licensing

It may 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.

- all future releases will be numbered ‘x.y.z’ (ie 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) will be 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) will be converted first into a request for the **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) will be converted into a request for the **links/qgraf-x.y.tgz** with the highest y such that **qgraf-x.y** has been declared ‘stable’;
- other requests, including those for a specific version, will not be converted (irrespective of availability);
- automatic downloads should point to the directory **links** instead of ‘vx.y’, eg


```
wget --quiet --user=anonymous --password=anonymous -O ./qgraf.tgz \
  http://qgraf.tecnico.ulisboa.pt/links/qgraf-3.4.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 the (any) downloading method openly in some webpage, ready for ‘bot use’ (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 *Downloads* webpage might still be temporarily available in the above mentioned **links** directory. At the time of the release of **qgraf-3.6.10** the convertible requests and the targets they point to 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-4.0.4.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-4.0.4.tgz
```

26.1 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 abilities 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, and which includes all (patch) versions **qgraf-x.y.z** with fixed **x** and **y**. Since declaring a version as stable the moment it is released does not seem to be the best strategy, there will be a delay (officially, a minimum of 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 anomalous behaviour, or error, and even then in a minimal way.

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 say 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 QGRAF*. If the package dependence is introduced⁷ at a time when QGRAF’s 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.

There is now a provision for distributing officially unavailable, stable versions⁸ of QGRAF in two cases (see the header of one of the latest Fortran files for details), provided it is legal to do so: (i) with (old) packages for which no upgrade is feasible, or (ii) if/when QGRAF’s official website closes down. Nevertheless, in the former case this provision might not be needed in practice as there is little backwards-incompatibility in **qgraf-3**, specially at the level typically required by other packages; in addition, I might be able to provide some help on such matters, should that kind of help be requested.

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).

⁷ The relevant date is the release date of that package version, obviously.

⁸ This means the latest patch version, of course; for example, **qgraf-3.4** is stable, but the actual version that would be distributed is (at present) **qgraf-3.4.2**.

27. Changelog for qgraf-3

3.1 (May 2005)

New features: the `vsum` and `psum` statements; the output displayed is more detailed.

3.1.1 (April 2008)

Fixes a problem with `psum`, as well as a minor bug related to the output displayed.

3.1.2 (September 2010)

Fixes a problem with `vsum`.

3.1.3 (November 2011)

Removes non-standard options of the `OPEN` statements from the source code (following a suggestion made by the `GoSam` collaboration) to make it automatically compatible with `gfortran`.

3.1.4 (October 2012)

New features: ability to read files containing ASCII text prepended by an UTF-8 byte order mark.

3.1.5 (February 2018)

Includes a minimal, partial ‘fix’ to one of the performance bottlenecks, which nevertheless provides, in some cases, the ability to generate diagrams at roughly one additional order of perturbation theory.

3.2 (June 2018)

New features: option `onshellx` and its dual; option `notfloop` (the dual of `floop`). Modifies the behaviour of option `nosnail` (and its dual) for 1-point diagrams. Fixes a problem with the style-file processing; fixes a problem with the diagram index (there is an error if the number of diagrams is *large*, ie $> 8 \cdot 10^8$), and allows that index to exceed 2^{31} . Minor optimizations.

3.3 (July 2018)

New features: the `index_offset` statement; option `bipart` and its dual; the `elink` and `plink` statements. Improves treatment of duplicate vertices. Fixes a problem with the model-file processing (involving the model-constants).

3.4 (January 2019)

New features: the `config` statement; options `cycli`, `onevi`, and their duals. Fixes a problem with option `nosigma`, and one other problem with the output displayed which involves the propagator types.

3.4.1 (March 2019)

Fixes a problem with the `plink` statement, and a problem involving the simultaneous use of options `nosnail` and `notadpole`. Adds adjective *connected* to the output displayed, to stress the fact that non-connected diagrams are not generated.

3.4.2 (April 2019)

Fixes a bug that generates problematic error messages about model-functions.

3.5.0 (May 2021)

First version compatible with the Fortran 2008 standard. New features: the `noblanks` config option; the `messages` statement; the ability to generate multiple output-files in the same run; options `noselfloop`, `nodiloop`, `noparallel`, and their duals. Improved memory management. Fixes a problem with `qgraf-3.4` (including `3.4.*`), which fails to detect some forbidden characters (eg `Tab`) in input files.

3.5.1 (March 2022)

New features: version reporting, with the command-line option `--version` (or `-version`). Fixes the text of an error message.

3.5.2 (June 2022)

Bug fix: enforces the computation of the internal momenta in some additional multiple-output instances. Announces ability to define the name of the control-file as a command-line argument.

3.5.3 (June 2024)

Bug fix: corrects a problem with `vsum`, which may occur in the presence of duplicate vertices.

3.6.0 (April 2022)

New features: a revamped input-model description; the `partition` statement; the keywords `<new_partition>`, `<new_topology>`, `<new_elinks>`, `<full_time>`, and `<raw_time>`. Introduces some algorithmic improvements into `vsum` and `psum`, to address certain special cases. As a proof of concept, it introduces a formal way to pass the amplitudes directly to another Fortran (and in the future, C) computer program, by using `QGRAF` as a subprogram.

3.6.1 (April 2022)

Fixes a bug involving the constants of the input model.

3.6.2 (May 2022)

Fixes a bug related to the decomposability of the input model and which may lead the program to bypass the diagram generation.

3.6.3 (June 2022)

New features: an extended `partition` statement. Bug fixes: restores ability to define the external momenta; enforces the computation of the internal momenta in some additional multiple-output instances; allows relative reordering of the `partition` and `index_offset` statements.

3.6.4 (September 2022)

New features: extended `loops` statement (and corresponding screen output); the keyword `<new_loops>`; the keyword `internal`. Bug fixes: corrects a problem with a recent modification of `psum`, which may result in the rejection of valid diagrams.

Removes the proof of concept feature (interface to Fortran programs) in anticipation of the expected assignment of a *stable* status to `qgraf-3.6`.

3.6.5 (October 2022)

Bug fixes: corrects a problem with the (recently extended) `loops` statement, which may disrupt some diagram options.

3.6.6 (May 2023)

Bug fixes: corrects an internal check that may lead the program to exit prematurely (with error code `trm_1`). A few other (very minor) modifications have been made, but it is not clear whether these include error fixes; for some compilers, or for some versions of some compilers, that might be the case (eg some previous statements may not be fully standard).

3.6.7 (December 2023)

Bug fixes: reinstates ability to use (i) unencoded rationals, in model-files, and (ii) strings with apostrophes (even before encoding, that is), in model-files and control-files; no longer displays some fragmentary information when the config option `noinfo` is used.

3.6.8 (June 2024)

Bug fixes: corrects a problem that occurs when the model includes vertices of degree larger than 10 (nevertheless, note that that degree range is clearly outside the default range).

3.6.9 (June 2024)

Bug fixes: corrects a problem with `vsum`, which may occur in the presence of duplicate vertices.

3.6.10 (July 2024)

Bug fixes: some non-decomposable models could be deemed decomposable.