

An introduction to QGRAF 3.0

P. Nogueira¹

CFIF, Edifício Ciência (Física), Instituto Superior Técnico,
1049-001 Lisboa, Portugal.

Contents

1. Intro
2. Strings, identifiers, and the such
3. Model configuration
4. The input statements
5. Intrinsic representation of diagrams
6. Output control I
7. Output control II
8. The screen output
9. Symmetric theories
10. Installation
11. Update
12. Final comments

¹ Researcher of the Technical University of Lisbon

1. Intro

QGRAF is a computer program for generating Feynman diagrams — more precisely, symbolic descriptions of Feynman diagrams — in quantum field theories. Its output, which the user can control in a number of ways, consists of a list of diagrams that match the input constraints; the sign — the one which follows from anti-commutation relations — and the symmetry factor of every diagram are also provided. **QGRAF** does not perform any other field theoretic calculation, however.

This program was created to automate the often tedious task of writing down Feynman diagrams and their symbolic amplitudes, especially when the number of Feynman diagrams is large. **QGRAF** is mainly directed at writing the amplitudes in terms of products of the matrix elements that follow from the Feynman rules, with no further processing. By presenting the symbolic amplitudes in a raw form it provides a common starting point for different types of calculations.

In the following sections we will try to describe in some detail how to use this program. A general description of the algorithm may be found in ref. [1]. Please note that the current version of the program does not generate vacuum diagrams, ie diagrams with no external fields.

2. Strings, identifiers, and the such

Let us start with a discussion on some very basic points regarding the input to be read by **QGRAF**. The alphabet to be used explicitly in the records read by the program is a subset of ASCII, namely the subset which contains the printable characters with codes ranging from 32 (blank) to 126 (tilde). This includes all the letters from A to Z (both lowercase and uppercase), the digits 0 through 9, parenthesis, braces, and several more symbols. Note in particular that ASCII character 9 (tab) is excluded. Some additional characters could be added to that alphabet — like the characters used to mark the end of the record/line, or the end of the file — with the understanding that they are control characters and their use occurs at a different level (the exact list even depends on the operating system or file system used). Hereafter we will often do our best to pretend they are not there.

Some types of strings are more relevant than others, so let us single out the ones that will be referred to later on. An *identifier* is a string made of letters, digits, and underscores, with the condition that the first character is a letter. Upper and lower case are not equivalent. The following strings provide three examples of identifiers.

spin F_0 csi__

Depending on what the output of **QGRAF** is used for, a more restricted class of identifiers may have to be used; for example, if the output is to be processed by a computer algebra system the identifiers used by **QGRAF** will have to be recognized as such by that program.

An integer is a sequence of digits, possibly preceded by a plus or minus sign: for instance, -0, +17, and 1234567 are integers. A rational is either an integer or a sequential concatenation of (a) an integer, (b) ASCII character 47 (slash), and (c) a nonzero unsigned integer; some examples are +0/5 and 3/7.

Note that the characters mentioned in any of the above definitions (for identifiers, integers, and rationals) are the only ones allowed. In particular, blank spaces are not allowed in those strings, although they are permitted in a different type of strings that will be described in the next paragraphs. When dealing with strings containing blanks it is often convenient to use the symbol \square — the so called visible space [2]. This symbol allows one to clarify whenever a blank is indeed part of a string, or where a record does begin, or end (if there are leading and/or trailing blanks).

Let us now specify a way to build strings that are accepted as indivisible by **QGRAF**, not parsed in the usual way, even if those strings contain several components such as identifiers, integers, punctuation marks, etc. For example, suppose one wants the program to accept the string

Quantum \square Electrodynamics \square in \square D=4-epsilon \square dimensions

as a single object. A possible solution to that problem consists in encoding the desired string into another one that cannot be confused with the other types of strings (identifiers, integers, rationals) and then supply the encoded string to **QGRAF** — which will simply decode it.

The right quote (ASCII character 39, also known as apostrophe) plays a special role in the encoding algorithm that was implemented. To encode a string is straightforward: first duplicate every right quote (ie replace each such character by two consecutive ones) and then add a right quote to each end of the resulting string. The encoded form of the above string is simply

‘Quantum \square Electrodynamics \square in \square D=4-epsilon \square dimensions’

Here are other examples: the empty string is encoded by the length 2 string ''; a string consisting of a right quote is encoded into the length 4 string '''. This type of encoding will be referred to by the designation *normal-encoding*. This encoding is exactly the one used in FORTRAN source programs, as it is well known (note, however, that the syntax for string concatenation is different — see below).

Records read by the program are limited to 80 printable characters, at least if the internal parameters are not changed. This condition places an effective bound on eg the size of identifiers, since an identifier cannot be broken into components. However, strings that are presented to the program in encoded form can escape that limit: before being encoded, any such string may be decomposed into several components, and then the encoded strings corresponding to those components can be written sequentially, either on the same line (separated by at least one blank!) or on consecutive lines (more on this later). For example, the string

$$'f(x)=' \cup \cup '1+\sin(x)'$$

is not the normal-encoding of any (single) string, but it will be interpreted by the program as a representation of the string

$$f(x)=1+\sin(x)$$

since the concatenation is implicit. We will say that a string t is an encoding of another string s iff t will be decoded as s , irrespectively of whether the encoding is normal or not. Hence a string may have numerous encodings.

3. Model configuration

3.1 The minimal description

Every model should be described in a file (hereafter called the model file) to be supplied by the user. The model file is formally divided into three sections, but the first of those is optional and will be described later. The first required section contains the declarations for the propagators and the last section contains the vertex declarations. For example, in the case of quantum electrodynamics the model file may look like this:

```
%_constants
%_propagators
_ [electron, _positron, _-]
_ [photon, _photon, _+]
%_electromagnetic_vertex
_ [positron, electron, photon]
```

Lines that start with any of the characters # (hash mark), % (percent sign), or * (asterisk) are ignored by the program and thus they can be used to write comments. The blanks that are present in the model file (and are not part of comment lines) serve mainly two purposes: to represent real blanks in encoded strings, or simply to isolate strings from one another. Frequently, they are not even needed for the latter purpose since other punctuation marks already fulfil that role.

Fields (or particles) are denoted by identifiers. In the above example there are three fields: `electron`, `positron`, and `photon`. There are two propagators and one vertex.

The basic syntax for the declaration of a propagator is

$$[\text{part}_1, \text{part}_2, s],$$

where `part_1` and `part_2` are fields and `s` is the sign of the propagator ('plus' for *bosonic* fields, ie fields satisfying commutation relations, and 'minus' for *fermionic* fields). The field `part_1` is the conjugate of `part_2`, and vice-versa; if `part_1` and `part_2` are the same field then we have a self-conjugate field. As it happens with ghost fields, sometimes `part_1` and `part_2` are not really particle and anti-particle; nevertheless `part_1` and `part_2` will be called (respectively) the *particle* and the *anti-particle*, in an absolute sense. What we call propagator represents a non-trivial contraction — ie vacuum expectation value of the time ordered product — of a pair of free fields. Graphically, it is simply a type of (possibly oriented) edge.

The basic syntax for the declaration of an interaction vertex of degree `n` is

$$[\text{part}_1, \text{part}_2, \dots, \text{part}_n].$$

Interactions are usually cubic or quartic, but in rare situations (eg exotic gauges, effective models) there will be interactions of higher degree. `QGRAF` will accommodate for this (degrees

in the range 3–6 are accepted by default, higher values require changing the value of a parameter in the source code). Some thought should be given to the ordering of the fields in the vertex declarations, making sure that each such declaration is consistent with the respective Feynman rule. For instance, when anti-commuting fields are present — as in the above example — one should not write `[electron,positron,photon]` if one has a Feynman rule that applies to the ordering $\bar{\psi}\psi A_\mu$, or else wrong signs may appear. Similar considerations apply to the propagator: one should use `[electron,positron,-]` to define the propagator $\langle \psi \bar{\psi} \rangle$.

Different declarations should be written on different lines. Declarations can extend across several consecutive lines provided the line breaks are consistent (this means that every ‘component’ — identifier, number, etc — must be contained wholly in a single line, and neither blank lines nor comment lines should be present in the middle of a declaration).

3.2 More parameters

The syntax declaration presented above is — let us stress it — the basic, minimal syntax. It lets us set the combinatorial description of the model, that is, what kind of lines there are and how they are allowed to meet at the nodes of the graphs (as well as whether the fields follow commuting or anti-commuting relations). That syntax must be extended to provide the ability to define parameters like mass, spin, charge, or even more complex objects representing Feynman rules, form factors, etc.

QGRAF allows users to define functions for fields, propagators, and vertices. Each such function — which maps either fields, propagators, or vertices to character strings — is represented by its own (freely chosen) identifier. If a function f has a finite domain then it can be specified by means of a finite number of assignments of the form $x \rightarrow f(x)$ — not unlike a tabular definition — without using a generic ‘formula’. That is the approach used in the program: to define a vertex function in the model file we simply have to state, in the declaration of every vertex, the image of the vertex under that function (see below for the exact notation). The same goes for propagator functions (just replace the word ‘vertex’ by ‘propagator’). Defining a field function is just slightly less trivial: every propagator declaration should contain either a single image or a pair of images (according to number of distinct fields that compose the propagator). Those images should be declared on the right hand side of the propagator and/or vertex declarations, which is separated from the left hand side by a semicolon.

A basis for the discussion of the extended syntax can be obtained from a suitable modification of the model file presented in the preceding subsection. Here is the new version, this time without visible spaces:

```
% constants

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

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

In this example the function `C` maps `electron` to `-1`, `positron` to `+1`, and `photon` to `0` (one may think of `C` as the electric charge); the function `gpow` maps the single vertex to `1` (this could be the power of the coupling constant appearing in the vertex Feynman rule); finally, `m` (this could stand for the mass) maps the fermionic propagator to `me` and the bosonic propagator to `m0`.

The notation for declaring the images of vertices and/or propagators is

```
function_id = S,
```

where `function_id` is the function identifier and `S` is the image (an encoded string). In the case of field functions the notation is either

```
function_id = ( S_1 , S_2 ),
```

if the particle is different from the anti-particle, or

```
function_id = ( S )
```

in the opposite case (`S`, `S_1`, and `S_2` denote encoded strings). It is clear that a propagator function is a special case of a field function, and can always be rewritten as such.

A simplification that is allowed in the definition of functions is the following: an image may be written unencoded whenever it is a valid identifier, integer, or rational (length allowing). Hence the previously presented model file can be simplified as follows.

```
% constants
% propagators
[electron, positron, - ; C= (-1, +1), m= me]
[photon, photon, +; C = (0 ), m= m0 ]
% electromagnetic vertex
[positron,electron,photon; gpow = 1 ]
```

Here are also some examples of declarations where that kind of simplification is not permitted: (1) `x = ''`, (2) `s = ' 1'`, (3) `sum= '2+1'`, (4) `key= 'a b'`, (5) `v = '(0)'`, (6) `rp = ''`.

3.3 Two optional keywords

There are two mutually exclusive keywords that can be used in the propagator declarations and which serve to further characterize the fields from the model file. The keyword `notadpole` is used to prevent the program from generating diagrams containing one-point functions of the field(s) declared in the statement where that keyword is used. If one modifies the previous declaration for the `photon` field and writes

```
[photon, photon, +, notadpole; C = (0 ), m= m0 ]
```

then diagrams with tadpoles of the field `photon` will be systematically suppressed.

The keyword `external` is used to specify that the field(s) to which it applies cannot be present as propagators of the diagrams generated by the program. In the following example

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

the field `Phi` is then a kind of external source (the program does not count external lines as propagators).

4. The input statements

4.1 The required statements

The basic instructions for QGRAF should be listed in the file `qgraf.dat`. The following example will help us discussing them in detail.

```
output = 'qlist' ;
style = 'qgraf.sty' ;
model = 'qed.' ;
in = electron, positron ;
out = photon, photon ;
loops = 2 ;
loop_momentum = k ;
options = onshell, floop;
```

Most of the structure of that file must be preserved; for example, those statements should not be removed, not even reordered. The statements no longer need to fit on a single line, and the restrictions for line breaking are exactly those that have been stated for the model file. Comment lines may also be present and are analogous to those of the model file. There are some optional statements (not listed above) that may be used to place further restrictions on the list of generated diagrams, but those will be discussed later.

Let us now analyze the eight required statements. The first statement declares the output file, in this case the file `qlist`. If this file already exists then the program will abort; for safety reasons, no overwriting is attempted. Specifying an empty filename (ie writing `output = ''`;) will tell the program not to generate the diagram list; it will still run and print a summary on the default output, usually the screen, but it will not create a list of diagrams. Unless one knows in advance that the number of diagrams is not too large, one should consider performing a first run without listing the diagrams in a file just to make sure the output will be of any use (and also to prevent filling up the computer disk).

The filename should be written in encoded form, as discussed in section 2. It should be noted, however, that any leading and/or trailing blanks in the filename will just be ignored. These rules apply also to filenames in other statements.

The second statement declares the style file, that is, the file that tells QGRAF how to present the diagram listing. The output style can be defined rather arbitrarily (see sections 'Output control I' and 'Output control II' for details).

The third statement declares the model file. Statements four and five define the incoming and outgoing fields, respectively. It is also possible to specify the external momenta. The statements

```
in = electron[p1], positron[p2] ;
out = photon[q1], photon[q2] ;
```

define both the external fields and their momenta: for example, `p1` is the momentum of the electron, flowing inwards, and `q1` is the momentum of the first photon, flowing outwards. If at least one momentum is declared, then all momenta must be declared. If no momentum

is declared the program uses the default momenta, which are defined as follows: \mathbf{p}_1 , \mathbf{p}_2 , \mathbf{p}_3 , etc, for the incoming fields and \mathbf{q}_1 , \mathbf{q}_2 , \mathbf{q}_3 , and so on, for the outgoing fields (respecting the order in which the fields were declared). Hence in the specific case of the above two statements, the momenta declarations could as well be omitted. The momenta should be identifiers; composite momenta like $-\mathbf{p}_1$, $\mathbf{p}_3-\mathbf{p}_1$, or $+\mathbf{k}$ cannot be used. Also, conflict with the integration momenta should be avoided.

The sixth statement specifies the number of loops of the diagrams. Statement seven defines the common prefix of the integration momenta associated to loops, which must be an identifier too. For instance, if this momentum prefix is equal to \mathbf{k} and the number of loops is equal to 2 then the two (symbolic) integration momenta will be \mathbf{k}_1 and \mathbf{k}_2 .

The last required statement allows one to specify a number of (mostly) topological properties that the Feynman diagrams should have. If no keywords are stated (`options = ;`) then all the connected diagrams satisfying the constraints imposed by the earlier statements will be generated. To make QGRAF discard certain types of diagrams (like 1-particle reducible diagrams, diagrams with tadpoles, etc) one simply has to list the appropriate keywords, separated by commas.

4.2 The optional keywords

Before describing the available options it is convenient to present some terminology. A 1-particle reducible diagram is one that can be disconnected by the removal of an internal edge; such an edge will be called a *bridge*. If some bridge carries zero momentum, no matter what the external momenta are — it is understood that the conservation of momentum is used at every vertex — it will be called a *singular* bridge; a bridge that is not singular is *regular*. A *tadpole* is part of a diagram connected to the rest of the diagram by a singular bridge. The diagram from fig. 1a contains a regular and a singular bridge (r and s , respectively); deleting s will clearly display the tadpole.

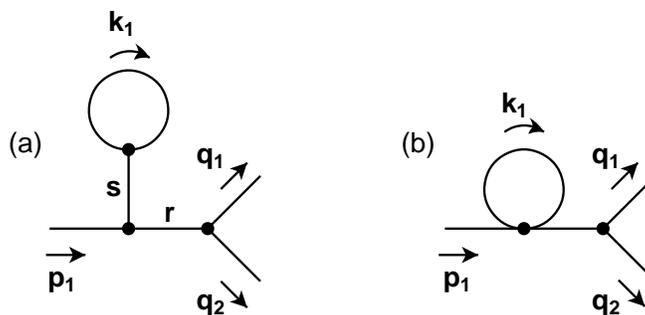


Fig. 1. Two snails but only one tadpole.

A *snail* is either a tadpole or a collapsed tadpole (ie one that can be obtained from a tadpole by eliminating the singular bridge defining the tadpole and merging the endnodes of that bridge into a single node). The diagrams in fig. 1 contain one snail each, the snail on the right being a collapsed version of the one on the left. These definitions apply even if nodes of higher degree are present.

Here is the list of the available options, together with a short explanation.

- `onepi` — 1-particle irreducible diagrams only

- `onshell` — no self-energy insertions on the external lines
- `nosigma` — no self-energy insertions (nowhere)
- `nosnail` — no snails
- `notadpole` — no tadpole insertions, ie no 1-point insertions
- `simple` — at most one propagator connecting any two different vertices, and no propagator starting and ending at the same vertex

The converse of these options are also available. Each of the following options

```

onepr
offshell
sigma
snail
tadpole
notsimple

```

rejects the diagrams validated by its counterpart, and vice-versa. It should be noted that all the above constraints are 'topological', that is, there is no reference to fields; it is not the whole diagram that matters, only the underlying graph. In addition, these options may be used in any combination, since they all eliminate some classes of diagrams.

Two other options are also allowed in special situations. When dealing with QED one may want to make use of Furry's theorem, and in that case the following option is available.

- `floop` — no graphs in which fermion loops have an odd number of interactions

As discussed in ref. [1], QGRAF generates diagrams by first generating a 'topology' (the underlying graph, consisting of nodes and edges), then finding all consistent ways of fitting the external fields and the interaction vertices, then proceeding to generate the next topology, etc. Consider now a different algorithmic flow: imagine that after finding a valid diagram the program would stop trying to find other diagrams with the same topology and would instead generate another topology. That would allow one to obtain an exact list of all the topologies that are present in the corresponding (complete) list of Feynman diagrams. The following option provides exactly that.

- `topol` — discards diagrams whose (unlabelled) topology is equal to that of an earlier generated diagram (to be used with a single neutral field, *only*)

4.3 The optional statements

Since the set of options just described is clearly insufficient other ways of selecting diagrams were implemented into the program. The generic form of those constraints is

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

where `<logical>` is either `true` or `false` and `<operator>` is one of the following keywords.

```

bridge
chord
iprop
rbridge
sbridge

```

The number of arguments k must be equal to or greater than 2, and the last two arguments — ie those with indices $k-1$ and k — should be non-negative integers, composed of at most four digits; furthermore, of those two arguments, the first should not exceed the second. The remaining arguments (if there are any) should be fields. Whenever present, those statements should come after the `options` statement.

To restrict the number of propagators of the field `phi` one may write eg

```
true = iprop[ phi ,3, 7] ;
```

This statement selects diagrams for which the number of propagators of the field `phi` is at least 3 and at most 7. If one replaces `true` by `false` then the diagrams selected are the ones in which the number of propagators of the field `phi` is either less than 3 or greater than 7. The role of the two numerical arguments is similar for all operators, only the quantity being constrained varies.

On other occasions one might be interested in propagators with certain topological properties. The operators `chord` (and `bridge`) enumerate propagators which belong (respectively, don't belong) to a loop. Bridges can be split into regular and singular ones, as explained, hence the operators `rbridge` and `sbridge`. For example, the statement

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

requires that there is at least one propagator of the field `photon` in a loop.

All these operators may have several fields as arguments. For instance, the statement

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

constrains the *sum* of the numbers of regular bridges with propagators of the fields `photon` and `electron`, respectively. In general, each field argument always contributes to the sum even if it is a duplicate; also, it does not matter whether a propagator is represented by the particle or the anti-particle. One may also have no field argument at all as in

```
false = bridge [ 1, 3] ;
```

in which case the total number of bridges is restricted.

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

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

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

The model file is intended as something fairly permanent, not something one should change temporarily for a single calculation and then change back again, especially if the same effect can be accomplished in the file `qgraf.dat`. However, both possibilities exist.

5. Intrinsic representation of diagrams

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

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

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

Let's see in more detail how one can define the embedding of fields, propagators, and vertices. Most of those cases are easy: propagators are attached to internal edges, vertices to internal nodes, and external fields to external nodes. What about the internal fields? Let us recall in the first place that in the perturbative expansion the interaction vertices supply the fields which are to be contracted in pairs, giving rise to propagators. Hence internal fields should be attached to objects surrounding the internal nodes. We could, for example, insert two auxiliary nodes into every internal edge and then attach the internal fields to those auxiliary nodes (as illustrated in fig. 3a). We don't need auxiliary nodes in the external edges — the external nodes will do.

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

5.1 The indices

If a diagram has V internal nodes and P internal edges then QGRAF numbers its vertices from 1 to V , and its propagators from 1 to P (see the examples given in figs. 2a and

2b). Those labellings define what we will call the vertex index and the propagator index, respectively.

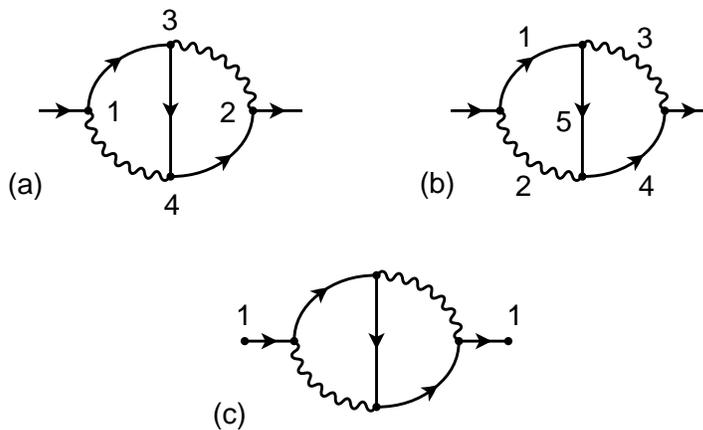


Fig. 2. Some indices for a simple diagram: (a) the vertex indices, (b) the propagator indices, and (c) the leg indices.

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

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

then the leg `positron` receives the in-index 1, the leg `electron` receives the in-index 2, the leg `higgs` receives the out-index 1, the leg `muon_minus` receives the out-index 2, and the leg `muon_plus` receives the out-index 3.

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

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

A related quantity is the *field type*. It takes only three values, namely 1 (for incoming fields), 2 (for outgoing fields), and 3 (for internal fields).

The sixth and last type of labelling will be called *ray index* because we may associate (in a visual sense) a propagator emerging from a vertex as a ray. For every vertex, the ray

index labels the surrounding vertex fields with the numbers $1, 2, \dots, D$ (D being the degree of the interaction), an example of which is given in fig. 3c. In contrast to other labellings, here labels differ only within each vertex; globally, there usually are repeated labels. The ray index is not always arbitrary: the index of an embedded field always coincides with the position (or one of the positions) of the field name in the definition of the interaction given in the model file. For instance, if an interaction has been defined as

`[positron,electron,photon]`

then, for vertices of this type, the field `positron` will always receive the ray index 1, the field `electron` will always receive the ray index 2, and the field `photon` will always receive the index 3, which means that in this case the labelling is unique (see fig. 3c). When the interaction contains repeated fields some arbitrariness remains.

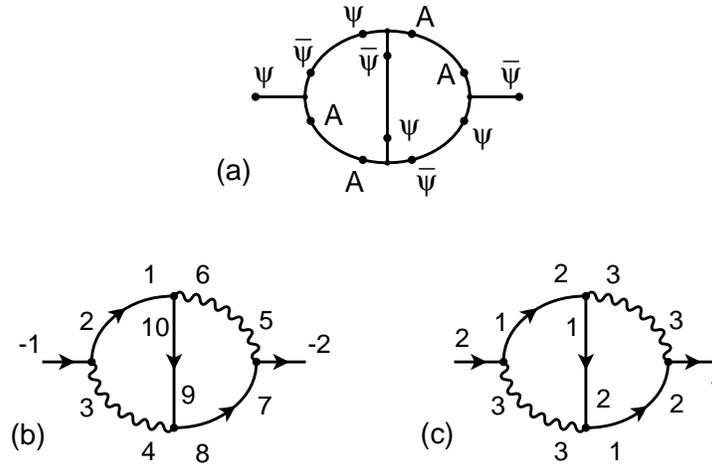


Fig. 3. Revisiting the diagram presented in fig. 2: (a) a way of embedding the internal fields using auxiliary nodes, (b) the field indices, and (c) the ray indices. The embedded fields are understood in both (b) and (c).

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

Note that some of the indices presented in this section — like the vertex index — are not completely determined in terms of a specific and complete rule: users cannot predict (relying on this manual, only) eg, the vertex indices of the vertices of a given diagram. Should the undocumented rules used internally by the program change in the future, no problems should appear provided the user assumes no special property for the indices other than the generic ones presented above.

5.2 The propagator orientation

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

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

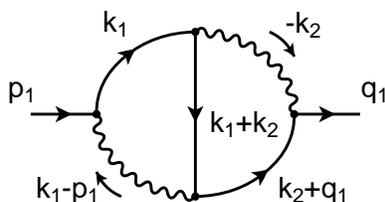


Fig. 4. The symbolic momenta for the diagram presented in figs. 2–3.

In fig. 4 one may see the symbolic momenta for the diagram presented in figs. 2–3. It may be verified at once, by taking a look at fig. 3b, that the orientation of the propagators (indicated with arrows) is in agreement with the above mentioned rule involving the field indices.

6. Output control I

In the first versions of **QGRAF** — before the release of version 2.0 — the list of diagrams in the output file would be generated according to a format selected from a short list of predefined output formats. Having a fixed number of predetermined formats is a self-limiting approach. Since every program that may be used to process the output of a diagram generator is most likely bound to have its own notation, the number of formats, and thus the number of subroutines, may have to be large. One alternative is that users write their own conversion subroutines, one for every program they think of using. Another problem with the fixed format approach is that it lessens the potential ability of the computer program to incorporate into the output the parameters that are part of the model definition (parameters for the fields, vertices, etc). If one wishes to have (i) a model file where the parameters are chosen by the user and (ii) an output containing a selection of those parameters, using a notation also chosen by the user, then the fixed format approach is inadequate.

In later versions there is a lot more flexibility, since the user has at his disposal a simple programming language to shape the output of the generator. It is not just a matter of choosing the type of delimiters, spacing and similar marks. Now users can also choose — the choice is limited, of course — what information they want to have on the output. In practical terms, it goes like this: to get a new format a user has to provide a file (hereafter called style file) containing a rather simple *program*, and that is all there is to it. Users can have a collection of such files, and use different formats on different occasions. What we claim is that it is much easier to write such a file than a full conversion routine. It may be impossible to write a style file that formats the output exactly like one wants; however, it should be possible to write a style file in such a way that the output file can be processed directly by one's favourite computer algebra system.

In simple terms, the output of **QGRAF** is as follows. At the beginning of the file (here we are discussing the output that is sent to a file) there is a prologue that may contain, for example,

- the name and the version of the program that generated the file,
- information required to identify the type of diagrams contained in the file (ie the statements found in the file `qgraf.dat`),
- extra information supplied by the user to communicate with to another program that will read the file (eg special marks to signal the beginning of the diagram listing).

After that the diagrams are listed one by one, following a general pattern. Finally there is a set of lines/records that usually serve to mark the end of the listing and/or the end of the file (for example, it is important to know that **QGRAF** has generated all the diagrams, and did not abort before completing the task). The characteristics of those three output sections are specified in the style file.

The syntax of the style file is completely different from any syntax discussed so far. Apart from comments — see below — that file contains 'text' (printable **ASCII** characters, that are taken literally) as well as keywords. A keyword is a reserved string that starts with the character `<` and ends with the character `>`, eg `<end>`. The four keywords that delimit the specification of the output sections, and which may be found in any of the style files provided with the program (the files whose name ends in `.sty`), are the following.

```

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

```

Those keywords should always come in that specific order. Each one of those keywords must be left-aligned on its own line, one containing no further information. Those four keywords are the only ones that must appear in any style file, and only once (note that a string like `<exit>` is not a keyword if it is part of a comment). Apart from syntax requirements all other keywords are optional — although a style file containing no keyword other than those required keywords is not terribly useful.

The prologue specification starts on the line following the `<prologue>` keyword and ends on the line that precedes the `<diagram>` keyword. Analogously, the diagram section specification is bounded by lines containing the keywords `<diagram>` and `<epilogue>`, and the epilogue specification comes between the lines defined by the keywords `<epilogue>` and `<exit>`. What comes either before the keyword `<prologue>` or after the line containing the keyword `<exit>`, if anything, must consist of blank lines or commentary (ie lines starting with the characters already defined for the other input files). In contrast, in the core part of the style file — that consists of the prologue, diagram section, and epilogue specifications — no comments are allowed and all the lines are taken as part of the output specification.

QGRAF reads the style file before starting generating diagrams, and it stores internally what it read. Just before it writes on the output file (it may do this many times) the program builds a string, representing a number of lines or records, by a sequence of two basic operations: appending one or more characters to the right end of the current string or deleting its rightmost character. The style file contains the instructions to build each such string.

6.1 The prologue section

For the sake of illustration let us will consider an imaginary style file, one whose prologue sections is defined by the following lines.

```

#
#_file_generated_by_<program>
#
<command_loop>#_<command_data><end><back>
#
_<back>
_tsum:=_0

```

This example contains most of the keywords that may be used in the prologue definition, and is interpreted as follows. The program starts with an empty output string. Then line 1 tells it to add a `#` as well as a `newline` character, which is always inserted when the end of a line is reached (to keep the discussion as simple as possible, we will pretend that there is a `newline` character whose purpose is to mark the end of a line and implicitly the beginning of a new line, whenever there is one). Line 2 tells the program to add another 20

characters, then to perform the action corresponding to the keyword `<program>`, and finally to add another `newline` character. The keyword `<program>` adds a string which contains the program's name and version number. What that string is may be deduced from the prologue below, which shows the actual output generated from the above specification (and from a file `qgraf.dat` that may also be guessed from the same output).

```
#
#_file_generated_by_qgraf-3.0
#
#_output_='qlist'_;
#_style_='sum.sty'_;
#_library_='';
#_model_='qed'_;
#_in =_electron_;
#_out=_electron,_photon_;
#_loops_=3;
#_loop_momentum_=k;
#_options_=floop,_onepi_;
#
_
_tsum_=0
```

Note that trailing blanks are ignored, no matter what type of input file is being read (there is usually some difficulty for a user to see them and, in addition, there is also some difficulty for `FORTRAN` to read them). Line 4 from the prologue specification is more interesting: the keyword `<command_loop>` tells the program to perform a loop (ie a programming loop): whatever is in between that keyword and the keyword `<end>` is executed a number of times, once for every statement found in the file `qgraf.dat`. The keyword `<command_data>` tells the program to insert the input statements in an orderly way, one statement each time the loop is executed. Note that a `newline` character is always part of every input statement (defining the end of the statement) and thus, in the above example, every iteration of the loop starts writing on a new line.

The keyword `<back>` deletes the rightmost character from the output string that is being built (this may generate an error if the string is temporarily empty). In the above example there are two occurrences of that keyword: the first one serves to delete the `newline` character from the last input statement, in order not to have two consecutive `newline` characters; in the second instance the purpose is to delete the second blank character on line 6, leaving a single blank (here is a way of generating trailing blanks). The keyword `<back>` may also be used to concatenate lines from the style file. This may be useful if one wishes to split a long line into two or more input lines. One reason for doing this is that the program assumes that input lines, from whatever input file, contain at most 80 (printable) characters. For example, the two input lines

```
This_is_a
<back>_single_line!
```

will be concatenated, since `<back>` will delete the `newline` character separating them (as long as the character `<` is in column 1). The use of `<back>` requires the program not to write a line as soon as it finds a `newline` character; instead, the output string is not written until the prologue specification has been fully executed (and similarly for other output sections). The keyword `<back>` may appear in any of the output sections; however, there can be no

interference among different sections (nor, in the case of the diagram section, interference between different diagrams).

Let us consider once more the previous prologue specification and the corresponding output. A simple examination of the output reveals that each command from the file `qgraf.dat` occupies a single line. However, if one rewrites that file so that at least one statement occupies more than a single line then the output may include something like this:

```
#_output
_=_
_ _ _ 'qlist' _;
```

That may or may not be what one wishes to obtain. QGRAF offers another possibility: one may use another kind of loop defined by the keyword `<command_line_loop>`, nested inside the loop defined by `<command_loop>`, eg,

```
<command_loop><command_line_loop>#_<command_data><end><back>
#
<end>#
```

The number of times that the inner loop is executed is equal to the number of lines in the input statement being addressed (which depends on the iteration of the outer loop), and the keyword `<command_data>` is now iteratively replaced by a single line of that statement. The latter example provides a way to have all output lines beginning with a hash sign, even if some input statements extend across two or more lines; in addition, it provides a separating line between different commands. Here is a fragment of the corresponding output:

```
#_output
#_ _ _ =
#_ _ _ _ 'qlist' _;
#
#_ _ style_ = _ 'sum.sty' _;
#
```

It should be noted that the keyword `<command_data>` is never replaced by empty lines and/or comments, only commands or lines that are part of a command, depending on the type of loop.

To present still another example, here is one specification that will format statements occupying more than one line into a single line:

```
<command_loop>#_<command_line_loop><command_data><back><end>
<end>#
```

There are five exceptional cases for representing characters in the style file. The first case is that of a blank (space character), if it is a trailing blank (see above). The other four cases are as follows (on the left hand side there are the ASCII characters one wishes to represent, and on the right hand side are the respective encodings).

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

This means that those four characters must be duplicated in the style file if they are to appear in the output file. With the help of this convention one may always tell ‘text’ and keywords apart. The first two of those exceptions are due to the use of the characters < and > in the delimitation of intrinsic keywords. The square brackets play a similar role in the case of keywords defined by the user.

6.2 The epilogue section

The epilogue section may contain all the keywords allowed in the prologue section, plus an additional one discussed below. This means that all the information about the input commands and the version of the program may be printed in the epilogue section instead, or even in both sections.

The additional keyword is <diagram_index>. As it will be seen in the next subsection, that keyword is mainly directed at the diagram section, where it instructs the program to produce a string representing the number of diagrams generated so far, including the current diagram. By a simple extension, in the epilogue section that keyword will produce a string representing the total number of diagrams generated by the computer program in that run.

6.3 The diagram section

The keywords may be divided into two main classes. One class contains what one may call control keywords; they serve to delimit the output sections, to define the programming loops, etc, but do not generate information by themselves. In this class we may find keywords like <diagram>, <command_loop>, <end>, and <back>. A second class contains the keywords that instruct the program to append information to the output string; these will be called data keywords. Up to now we have already seen three data keywords, namely <program>, <command_data>, and <diagram_index>, but many more exist.

Data keywords may themselves be divided into local and global keywords. Local keywords are those that must be used inside one of the programming loops accepted in the style file, while global keywords have no such restriction. Global keywords don’t have to be constants, for example the keyword <diagram_index> will produce different strings at different stages.

We will now discuss the diagram section, which is obviously the most important. There are many keywords that may be used in that section, most of which are data keywords. The global keywords are listed below.

- <diagram_index> — a positive integer specifying the order in which a diagram was generated (ie 1 for the first diagram, 2 for the second diagram, etc)
- <legs> — the number of external fields of the diagram
- <legs_in> — the number of incoming fields
- <legs_out> — the number of outgoing fields
- <loops> — the number of loops of the diagram

- `<minus>` — similar to `<sign>` (see below) if the diagram sign is minus, otherwise it produces an empty string
- `<propagators>` — the number of internal edges of the diagram
- `<sign>` — the diagram sign (either a plus or a minus sign) that follows from the anti-commutation rules
- `<symmetry_factor>` — the diagram symmetry factor (either 1, if there are no symmetries, or a fraction like 1/2, 1/6, etc)
- `<symmetry_number>` — the diagram symmetry number (a positive integer equal to the reciprocal of the symmetry factor)
- `<vertices>` — the number of internal nodes of the diagram

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

```

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

```

announce four of those loops — to be called, respectively, incoming loop, outgoing loop, propagator loop, and vertex loop — and the keyword `<end>` closes them. Those loops are executed as many times as there are (respectively) incoming particles, outgoing particles, propagators, and vertices in the diagram being listed. During the execution of such a loop the program prepares itself to inspect the relevant class of objects (ie legs, propagators, or vertices) and prints information about them if it is requested to do so.

The fifth loop is defined by the keyword `<ray_loop>` and it should always appear nested inside the vertex loop, like this:

```

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

```

The ray loop is needed to tell the program to inspect every line incident with the vertex that is implicitly defined by the vertex loop.

Those five programming loops form the basic tool to access the local information that defines a diagram. By local information we mean that it refers either to the component of the diagram being examined, or to some neighbouring component (as opposed to a component that is on a remote part of the diagram, so to say). For instance, if the object being inspected is a vertex v_0 then some information regarding the vertices adjacent to v_0 is also available at that moment, as is the information on the propagators (or external lines) incident with v_0 . However, at the time that v_0 is inspected no information about more remote objects is available, with the obvious exception of the information provided by the global keywords presented above.

What remains to be explained here is which (local) keywords may be used inside which loops, as well as what the keywords stand for. The former of those issues is addressed in the table shown below. A given local keyword may be used in a certain loop type iff the respective table entry is marked with a full circle. The loop types are denoted by their initials — i for incoming loop, r for ray loop, etc. It is clear that if a keyword may be used in the vertex loop then it may also be used in the ray loop — although the information it represents will remain constant while the ray loop is executed.

	i-loop	o-loop	p-loop	r-loop	v-loop
<dual-field>	•	•	•	•	
<dual-field_index>			•	•	
<dual-momentum>	•	•	•	•	
<dual-ray_index>			•	•	
<dual-vertex_degree>			•	•	
<dual-vertex_index>			•	•	
<field>	•	•	•	•	
<field_index>	•	•	•	•	
<field_sign>	•	•	•	•	
<field_type>	•	•	•	•	
<in_index>	•				
<leg_index>		•			
<momentum>	•	•	•	•	
<out_index>		•			
<propagator_index>			•	•	
<ray_index>	•	•	•	•	
<vertex_degree>	•	•	•	•	•
<vertex_index>	•	•	•	•	•

The exact definition of a local keyword depends on the type of loop where it is used. We will now review the basic loop types and describe all those keywords.

6.3.1 The propagator loop

The propagator loop is executed as follows: when the program finds the keyword <propagator_loop> it assigns the value 1 to the corresponding loop index, preparing itself to examine the propagator whose propagator index is equal to 1. Then it continues execution, printing the information requested about that propagator until it finds the keyword <end>. At that point it increments the loop index to 2, and the rest you may guess. It only exits the loop when all the propagators have been visited. Let us recall that if a propagator has been assigned a propagator index i then the corresponding fields are Φ_{2i-1} and Φ_{2i} .

To have an idea of the information available during the execution of the propagator loop one may observe fig. 5, which has been obtained by grouping together figs. 2–4 but retaining only part of the original diagram, namely propagator 5 and its neighbourhood.

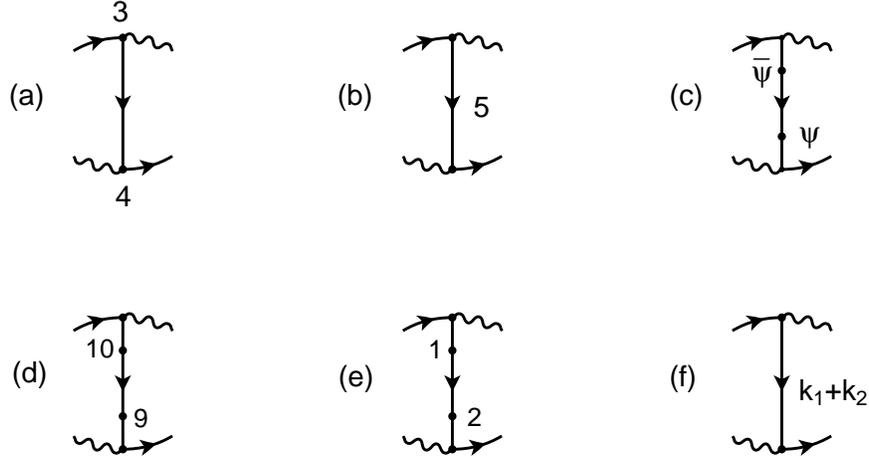


Fig. 5. Some local information available in the propagator loop: (a) vertex index, (b) propagator index, (c) propagator fields, (d) field index, (e) ray index, and (f) propagator momentum.

A list of the keywords allowed in the propagator loop is given below. Each entry contains a keyword, its meaning, and an output string generated by that same keyword. The output string is the one that would be obtained in the description of the propagator shown in fig. 5 (the string is in parenthesis, after the arrow symbol), in which case we may identify ψ with **electron** and $\bar{\psi}$ with **positron**.

- `<dual-field>` — the name of the field Φ_{2i} (\rightarrow **positron**)
- `<dual-field_index>` — the unsigned integer $2i$ (\rightarrow 10)
- `<dual-momentum>` — the symmetric of `<momentum>` (\rightarrow -k1-k2)
- `<dual-ray_index>` — the ray index of the field Φ_{2i} (\rightarrow 1)
- `<dual-vertex_degree>` — the degree of the interaction vertex to which the field Φ_{2i} belongs (\rightarrow 3)
- `<dual-vertex_index>` — the vertex index of the interaction vertex to which Φ_{2i} belongs (\rightarrow 3)
- `<field>` — the name of the field Φ_{2i-1} (\rightarrow **electron**)
- `<field_index>` — the unsigned integer $2i-1$ (\rightarrow 9)
- `<field_type>` — the field type of the field Φ_{2i-1} , which is always equal to 3 (\rightarrow 3)
- `<field_sign>` — the sign of propagator i (\rightarrow -)
- `<momentum>` — the momentum of propagator i (\rightarrow k1+k2)
- `<propagator_index>` — the unsigned integer i (\rightarrow 5)
- `<ray_index>` — the ray index of the field Φ_{2i-1} (\rightarrow 2)
- `<vertex_degree>` — the degree of the vertex that contains Φ_{2i-1} (\rightarrow 3)
- `<vertex_index>` — the vertex index of the vertex that contains Φ_{2i-1} (\rightarrow 4)

6.3.2 The leg loops

Let r (respectively, s) be the number of incoming (respectively, outgoing) particles. Those particles are listed in the file `qgraf.dat` in a certain order, and we will denote the n^{th} incoming (respectively, outgoing) particle by Φ_n^{in} (respectively, Φ_n^{out}).

As seen earlier, there are two types of leg loops, the incoming loop and the outgoing loop. In the former case the loop is executed r times, once for each incoming field, while in the latter case the loop is run s times, once for each outgoing field (the legs of the diagram are, rather obviously, the main objects that are accessed in those loops).

Let us consider first the incoming loop, where the loop counter i runs from 1 to r . Let v_k be the vertex that leg i is incident to, and j the ray index (with respect to that vertex) of the field Φ_i^{in} . The field Φ_i^{in} can then be identified with $\Phi_{k,j}$.

The same diagram that has been used before (figs. 2–4) is also used here as a basis for the discussion. Its legs are the ones specified by the following statements,

```
in = electron ;
out = electron ;
```

and thus the output presented with every keyword contains a single string (since $r=s=1$).

- `<dual-field>` — the conjugate of `<field>` (\rightarrow positron)
- `<dual-momentum>` — the symmetric of `<momentum>` (\rightarrow -p1)
- `<field>` — the name of Φ_i^{in} (\rightarrow electron)
- `<field_index>` — the field index of Φ_i^{in} (\rightarrow -1)
- `<field_type>` — the field type of Φ_i^{in} (\rightarrow 1)
- `<field_sign>` — the sign of Φ_i^{in} (\rightarrow -)
- `<in_index>` — the unsigned integer i (\rightarrow 1)
- `<momentum>` — the momentum flowing into the diagram through leg i (\rightarrow p1)
- `<ray_index>` — the unsigned integer j , which is the ray index of $\Phi_{k,j}$ (\rightarrow 2)
- `<vertex_degree>` — the vertex degree of v_k (\rightarrow 3)
- `<vertex_index>` — the vertex index of v_k , which is the unsigned integer k (\rightarrow 1)

If the loop type is `<out_loop>` some of the above definitions need to be changed. The loop counter i goes from 1 to s , but the leg index goes from $r+1$ to $r+s$. The keyword `<field>` now produces outgoing fields, and the momentum direction is the one leaving the diagram. Thus $\Phi_{k,j}$ is now the *conjugate* of Φ_i^{out} .

- `<dual-field>` — the conjugate of `<field>` (\rightarrow positron)
- `<dual-momentum>` — the symmetric of `<momentum>` (\rightarrow -q1)
- `<field>` — the name of Φ_i^{out} (\rightarrow electron)
- `<field_index>` — the field index of Φ_i^{out} (\rightarrow -2)
- `<field_type>` — the field type of Φ_i^{out} (\rightarrow 2)
- `<field_sign>` — the sign of Φ_i^{out} (\rightarrow -)

- `<leg_index>` — the unsigned integer $r+i$ ($\rightarrow 2$)
- `<momentum>` — the momentum leaving the diagram through leg i ($\rightarrow \mathbf{q1}$)
- `<out_index>` — the unsigned integer i ($\rightarrow 1$)
- `<ray_index>` — the unsigned integer j , which is the ray index of $\Phi_{k,j}$ ($\rightarrow 1$)
- `<vertex_degree>` — the vertex degree of v_k ($\rightarrow 3$)
- `<vertex_index>` — the vertex index of v_k , which is the unsigned integer k ($\rightarrow 2$)

6.3.3 The vertex and the ray loops

The vertex loop tells the program to visit the internal vertices of the diagram, and the ray loop that it should also visit every line incident with each such vertex. The two indices that control these loops are the vertex index and the ray index (in the following we will denote them by i and j , respectively).

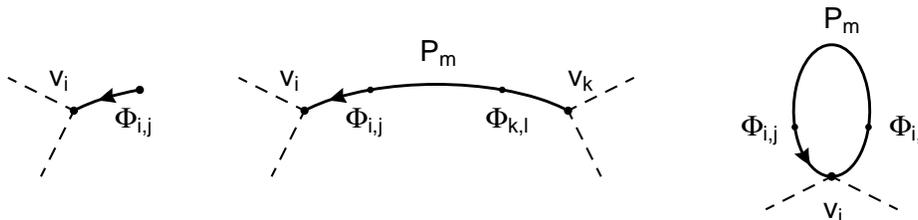


Fig. 6. Basic notation used in describing the vertex and ray loops.

A vertex v_i of degree d_i comprises the fields $\Phi_{i,j}$, for $j=1, \dots, d_i$. Fig. 6 illustrates the three possible cases for $\Phi_{i,j}$: it can be an external field (left), an internal field that is part of a propagator joining two different vertices (centre), or an internal field that is part of a propagator built from two fields of the same vertex (right). If, for a given value of j , $\Phi_{i,j}$ is an internal field then there is a propagator P_m connecting v_i to another vertex v_k (or else connecting v_i to itself, in which case we just set $k = i$); that propagator also contains another field — belonging to v_k — which is the conjugate of $\Phi_{i,j}$ and that will be denoted by $\Phi_{k,l}$. Observe that v_k and $\Phi_{k,l}$ are both undefined whenever $\Phi_{i,j}$ is an external field.

In the ray loop the keyword `<momentum>` refers to the momentum flowing across the edge to which $\Phi_{i,j}$ is attached, in the direction that is shown graphically in fig. 6 by means of arrows. Loosely, this graphical rule gives the momentum flowing into vertex v_i coming from that edge, except that if $\Phi_{i,j}$ is an internal field and $i = k$ then this wording must be made more precise.

Fig. 7 contains part of the Feynmann diagram presented earlier in this section — it shows vertex 1 and its neighbourhood — and will also be used for illustrating the meaning of the keywords presented below. This time the output of the program given with each entry may be divided into two cases. For the keywords that do not require the use of the ray loop construction the output string contains a single label, the one corresponding to vertex 1. For the other keywords the output string is composed of three sub-strings (since the degree of vertex 1 is equal to 3), one for each ray index; that means that the execution of the ray loop is simulated, but the execution of the vertex loop is not.

- `<dual-field>` — the name of the conjugate of $\Phi_{i,j}$ (\rightarrow electron positron photon)
- `<dual-field_index>` — the field index of $\Phi_{k,l}$ if that field exists, otherwise zero (\rightarrow 1 0 4)
- `<dual-momentum>` — the symmetric of `<momentum>` (\rightarrow k_1 $-p_1$ $-k_1+p_1$)
- `<dual-ray_index>` — the ray index of $\Phi_{k,l}$ (ie the unsigned integer l) if that field exists, else zero (\rightarrow 2 0 3)
- `<dual-vertex_degree>` — the degree of the interaction vertex v_k if it exists, else zero (\rightarrow 3 0 3)
- `<dual-vertex_index>` — the vertex index of v_k (ie the unsigned integer k) if it exists, otherwise zero (\rightarrow 3 0 4)
- `<field>` — the name of the field $\Phi_{i,j}$ (\rightarrow positron electron photon)
- `<field_index>` — the field index of $\Phi_{i,j}$ (\rightarrow 2 -1 3)
- `<field_type>` — the field type of $\Phi_{i,j}$ (\rightarrow 3 1 3)
- `<field_sign>` — the sign of $\Phi_{i,j}$ (\rightarrow - - +)
- `<momentum>` — the momentum flowing into vertex v_i coming from the edge to which $\Phi_{i,j}$ is attached, as explained before (\rightarrow $-k_1$ p_1 k_1-p_1)
- `<propagator_index>` — the propagator index of P_m (ie the unsigned integer m) if it exists, otherwise the field index of the external field $\Phi_{i,j}$ (\rightarrow 1 -1 2)
- `<ray_index>` — the unsigned integer j (\rightarrow 1 2 3)
- `<vertex_degree>` — the degree of vertex v_i (\rightarrow 3)
- `<vertex_index>` — the unsigned integer i (\rightarrow 1)

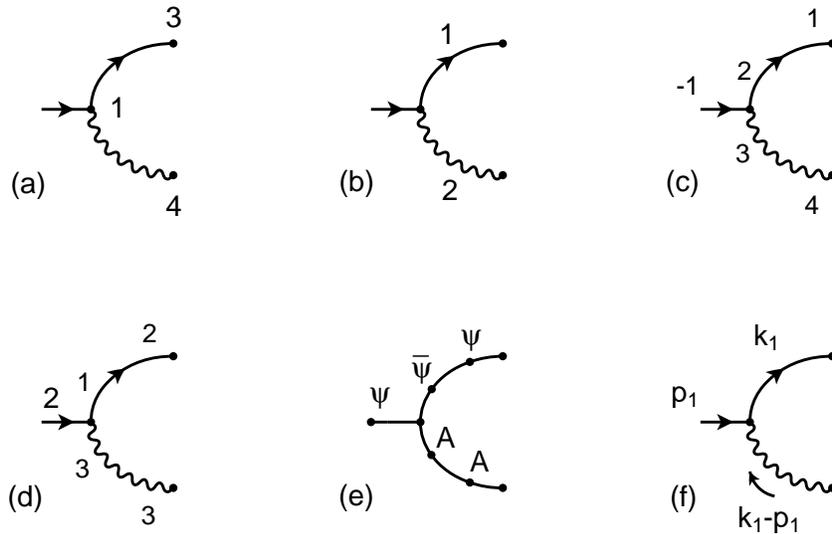


Fig. 7. Basic information available in the vertex loop.

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

6.4 The diagram sign

The diagram sign computed by QGRAF is equal to $(-1)^{T_A}$, with T_A being the number of transpositions of anti-commuting fields that are needed to bring those fields from their natural vertex ordering (defined in the model file) to the ordering defined by the propagator pairing plus an *ad hoc* reordering of the unpaired fields. The unpaired fields are ordered according to their leg indices, the outgoing fields to the left of the incoming fields, the former in increasing order and the latter in decreasing order:

$$(\Phi_1^{*out} \Phi_2^{*out} \dots \Phi_s^{*out}) (\Phi_r^{in} \dots \Phi_2^{in} \Phi_1^{in}).$$

As an example, let's take the diagram displayed in figs. 2–4, and assume the usual $\bar{\psi}\psi A_\mu$ ordering for the interaction vertex. Using the field indices given before, the vertex product can be written as

$$(\bar{\Psi}_2 \Psi_{-1} A_3) (\bar{\Psi}_{-2} \Psi_7 A_5) (\bar{\Psi}_{10} \Psi_1 A_6) (\bar{\Psi}_8 \Psi_9 A_4).$$

After discarding the commuting fields a simple reordering will lead to

$$+ (\bar{\Psi}_{-2}) (\Psi_{-1}) (\Psi_1 \bar{\Psi}_2) (\Psi_7 \bar{\Psi}_8) (\Psi_9 \bar{\Psi}_{10}).$$

Hence QGRAF will say that the sign of that diagram is $+$.

Users don't really have to use the diagram sign computed by QGRAF. As it should be clear by now, it is very simple to omit any reference to the diagram sign in the output file, and then users are free to implement their own definition.

6.5 A first survey of output styles

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

The main issue concerning the choice of output style is the processing of the expressions built by QGRAF. The output styles defined by the files `sum.sty`, `array.sty`, and `form.sty` may serve to illustrate three different approaches. The first one consists in combining all the symbolic expressions into a single expression, that is, to add them up. Further processing may be problematic if the number of diagrams is large. In the second approach an array/vector

is defined, each component storing the symbolic expression for a single diagram; then the processing program reads the whole array and manipulates all the expressions in a single run. In some cases this may still be problematic, or just inefficient. In the third approach, which has been used together with FORM [3], the expressions are still kept in a single file but each expression can be read and processed separately.

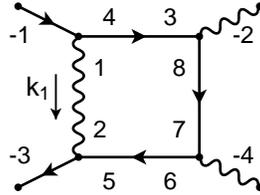


Fig. 8. A diagram from QED, and the corresponding field indices (the embedded fields are understood).

Let us now look at the output produced by the style files `array.sty` and `form.sty` that are part of release 3.0. Although the corresponding output styles are outdated, a brief analysis of those styles can still be profitable if it provides a basis for building other output styles. The examples presented below refer to the diagram shown in fig. 8. The first of those uses the file `array.sty`:

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

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

Using the file `form.sty` the same diagram is described as follows.

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

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

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

7. Output control II

In the previous section we discussed the output features that are available by default. However, as seen in the section “Model Configuration”, it is possible to define parameters for the fields, propagators and vertices of a model. The subject of the present section is then how to make use of such parameters, ie how to program the style file so that they appear in the output file.

7.1 Field, propagator, and vertex functions

Let us go back to QED, and consider the following model file:

```
% constants

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

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

We may interpret the function `pfunct` as representing the “propagator function” and `m` as the “propagator mass”. Coupling that model file with the following code (which is meant to be part of the diagram section of the style file)

```
<propagator_loop>□[pfunct](<momentum>, [m])*
<end>
```

leads, in the case of the diagram represented in fig. 4, to the following output:

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

In this elementary example we can see how to obtain an output where each propagator type has its own name, and its own mass. As seen earlier, implicitly defined functions (ie those that are defined within `QGRAF`) are denoted by strings enclosed in angle brackets (in fact, the symbols `<` and `>`). Now we may see that the functions defined by the user should have their names enclosed in *square brackets* when referred to in the style file. This prevents any interference between the two sets of functions, and so the user can define his functions by any valid identifier.

It is probably worth stressing that the type of functions considered in this subsection have as their domain a set of objects from the model file, not a set of embedded objects. In other words, when the computer program evaluates say, the result corresponding to the function represented by the string `[pfunct]` it only looks at the field names that appear in the propagator, not at the propagator index; hence `[pfunct]` gives the same result for all embedded propagators sharing the same fields (ie, field names).

As seen in the previous section, keywords like `<field>` and `<vertex_index>` can be used in more than one kind of loop provided one makes appropriate definitions in each case. In a parallel way one may accept the inclusion of user defined functions in more than one type of loop. This holds even for propagator functions, which are always seen as a special case of field functions. In some cases the user defined functions may be prefixed with the string `dual-`, just like some intrinsic functions.

Propagator functions are treated mostly like field functions in what concerns the output of the program; the only difference is that the prefix `dual-` will not be allowed. The following table shows which kind of functions are allowed in which type of loop (the presence of a full circle in a matrix entry denotes permission, and its absence interdiction).

	i/o-loop	p-loop	r-loop	v-loop
<code>[ff]</code>	•	•	•	
<code>[dual-ff]</code>	•	•	•	
<code>[pf]</code>	•	•	•	
<code>[dual-pf]</code>				
<code>[vf]</code>	•	•	•	•
<code>[dual-vf]</code>		•		

The strings `ff` and `vf` denote generic field and vertex functions, respectively; they map fields ϕ and vertices ν from the model into strings `ff(ϕ)` and `vf(ν)`. The string `pf` denotes a generic propagator function. The definitions of the functions listed in the above table depend on the type of loop they appear in, but it is possible to present unified definitions using a few intrinsic keywords that are loop dependent.

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

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

7.2 The constants

The discussion about the first part of the model file, which was omitted in section 3, is the subject of the present subsection. That part of the model file is the place where to define the constants of the model. That could be a name for the model, or some other string that should appear in the output file, even as a comment. Admittedly, the choice is rather limited; the usefulness of that part of the model file will hopefully increase in the future when non-constants are allowed to come into play.

The following statement shows a typical definition of a constant (note that any such statement should define a unique constant).

```
[_model=_ 'massless_QCD_in_D=4-eps_dimensions']
```

If the constant `model` is defined on a set of model files then it is possible to use the keyword `[model]` in (say) the prologue of a style file and specify in a more precise manner the model which is being considered instead of just guessing from the model filename (it is being assumed that there is a style file shared by various models).

Constants may be used in any section of the style file, and may be invoked just like the field/propagator/vertex functions, ie enclosing the name of the constant in square brackets.

8. The screen output

Even when there is no output file there is a minimal output sent to the default output logical unit, usually some window on the screen. That output consists mainly of the following information: (a) the version of the program, (b) the statements found in the file `qgraf.dat`, (c) the model partitions, (d) the possible vertex degree partitions and, for every such partition, the matching number of diagrams, and (e) the total number of diagrams found. Here is an example:

```

-----
                                qgraf-3.0
-----

output = 'qlist' ;
style = 'qgraf.sty' ;
library = '' ;
model = 'scalar' ;
in = H, H ;
out = H, H ;
loops = 2 ;
loop_momentum = k ;
options = onshell ;

-----

model partitions:  P^1  V^2

                   B^1  3^1  4^1

-----

-   4^3 --- 12 diagrams
3^2 4^2 --- 300 diagrams
3^4 4^1 --- 457 diagrams
3^6  - --- 153 diagrams

total = 922 diagrams

-----

```

The first model partition shows the number of propagators (the exponent of P) and the number of vertices (the exponent of V) found in the model file. The second partition is a refinement of the first, and displays the following numbers, if positive: the number of bosonic propagators (the exponent of B), the number of fermionic propagators (the exponent of F), and the numbers of vertices with degree 3, 4, etc (the number of vertices with degree k is the exponent of k in that expression). Terms corresponding to null values are omitted. The example we have just presented corresponds to a run where the model has a single

propagator (that of a real bosonic field) and two vertices (one cubic and one quartic self-interaction terms).

Let ν_k ($k = 3, 4, \dots$) be the number of vertices of degree k in some Feynman diagram with a given number of legs and number of loops. If the last two numbers are kept fixed and the ν_k are regarded as variables, then it should be clear that there is a finite number of possible solutions for the vector $\nu = (\nu_3, \nu_4, \dots)$. For example, there is no 1-loop propagator diagram having 8 cubic vertices. Each vertex degree partition is a possible solution for the vector ν (possible in the sense that it satisfies the basic partition-like equation) and is displayed using a common notation for partitions. If we go back to our previous example we will see that the program found 12 diagrams with three quartic vertices (and none cubic), 300 diagrams with two cubic vertices and two quartic ones, etc.

9. Symmetric theories

QGRAF is most suitable for *broken-symmetry* models. However, in the case of symmetric theories some extra work is usually wanted. For the purpose of illustration let us take QCD (6x3 quarks, 8 gluons and 2x8 ghost fields). It is clearly possible to obtain a list of all the graphs for a particular process provided that the model file lists *all* the interactions of that model, that is, all the vertices for all possible colour indices (there are roughly 200 vertices for QCD). This will result in a very large number of diagrams, which is not what one usually wants. One usually prefers a shorter list where each generic diagram represents all the diagrams of the larger list that differ only in the values of some colour indices. However, this approach requires the computation of algebraic factors (the so-called colour traces) to take into account the contributions of the diagrams represented by each generic diagram.

The shorter list may be obtained in a simple way, by not including colour indices (ie typing in the model as if there were eg 6 quarks, 1 gluon and one pair of ghosts). But then one should rely on other means to get the colour traces. If the diagrams are to be evaluated symbolically with the help of a computer then one may use a computer algebra program to have the colour trace computed. This is not a trivial task, specially if one wants to have a somewhat arbitrary gauge group [4].

10. Installation

QGRAF is almost entirely written in standard FORTRAN-77. Depending on your compiler, you may have to execute the following steps (before compiling the code) in order to eliminate the nonstandard features.

- adjust the OPEN statements (4 occurrences) to your system/compiler requirements; by default, those statements incorporate two nonstandard qualifiers (`readonly` and `carriagecontrol`) that work only with a restricted set of compilers. If you get an error message from your compiler, remove the incompatible qualifiers.
- change characters to uppercase if necessary.

Summarizing: adjust the code to your needs; compile, link, and then execute (the model file, the style file, and the file `qgraf.dat` should be provided). Please note that the program must be called from the directory where the file `qgraf.dat` is located. It should be a good idea trying to reproduce some of the numbers listed in ref. [1]. Please report any discrepancies.

As the program does not allocate memory dynamically, the size of the memory needed to store all the information supplied to and/or used by the program is controlled by means of parameters, and thus set when the program is compiled. When the default values of those parameters are insufficient for QGRAF to handle successfully a certain problem the program will abort. However, if one adjusts (ie increases by a sufficient amount) the values of the relevant parameters and recompiles the source code, the program will be able to run. A serious effort has been made with version 3.0 to reduce the number of parameters that the user might have to change. If a model contains many fields, vertices, functions, etc, then the following parameters may have to be increased.

- `sibuff` — the dimension of the integer array where most integer values are stored
- `scbuff` — the size of the character buffer where most strings are stored

Those parameters are set in the following statement

```
parameter ( sibuff=524288, scbuff=131072 ),
```

which appears in several routines.

The bounds on the number of loops and legs are as follows.

$$\begin{aligned} \text{loops} &\leq \text{maxrho} \\ 1 &\leq \text{legs} \leq \text{maxleg} \\ \text{legs} + \text{loops} &\leq \max(\text{maxrho}, \text{maxleg}) \\ 3 &\leq \text{legs} + 2*\text{loops} \end{aligned}$$

The default values for `maxrho` and `maxleg` are 5 and 9, respectively. Those parameters can be changed, but in practice that won't be needed often, if ever. Users requiring higher values should take extra precautions as the present cross checks on the program do not cover that domain. In addition, the implemented algorithm is not efficient for large values of `nloop`. A faster sub-algorithm is under study (low priority). In any case, the number of graphs tends to increase very rapidly with the number of loops — roughly in a factorial like way — and the average time to generate a single graph also increases somewhat.

11. Update

Version 3.0 represents the second stage towards the goal of providing a truly powerful built-in tool to shape the output of the program. This time, support for user-defined parameters has been included. Other significant improvements include: (a) a more flexible syntax in file `qgraf.dat` and in the model file, and (b) a redesign of the program in order to (i) reduce the number of parameters that may have to be changed by the user and (ii) ease the future implementation of better memory allocation techniques. That redesign required a substantial rewriting of the computer program.

There are several changes with respect to version 2.0 (see below for a list). Not every change may be classified as strictly necessary but, given that some changes were needed, a few others which could be helpful in some way (particular in clarifying the notation, or making it more uniform) were also implemented. The entries that are marked with an open circle are the ones that introduce incompatibilities — in the sense that input files valid for version 2.0 require (or may require) some modification before being accepted by version 3.0. Let us start with the changes related with the input file `qgraf.dat`:

- statements may be split across multiple lines
- option `notadp` has been replaced by `notadpole`
- some new options, mostly duals of the previously existing ones (eg `onepr`, `tadpole`)
- two new operators (`sbridge` and `rbridge`) allowed in the optional statements

The changes regarding the model file are:

- field/propagator/vertex parameters may be defined
- identifiers may be longer
- statements may be split across multiple lines
- vertices no longer have to be listed in order of ascending degree
- new keywords `external` and `notadpole` replace old notation (`p` and `t`) in propagator declarations

Finally, the changes for the style file:

- the parameters from the model file may be used in the output specification
- new keyword `<field_type>`
- new loops `<in_loop>` and `<out_loop>` replace `<leg_loop>`
- keyword `<sub_loop>` has been replaced by `<ray_loop>`
- keyword `<propagator_index>` substitutes `<edge_index>`
- keyword prefix `dual_` has been replaced by `dual-`
- `<command_loop>`, `<command_line_loop>` and `<command_data>` supersede keywords `<prologue_loop>` and `<data>`
- the keywords `<leg_field>` and `<leg_momentum>`, as well as their duals, are obsolete
- all the keywords accepted in the prologue section may be used in the epilogue section

12. Final comments

QGRAF is copyrighted software that may be freely used for academic purposes. As it should be clear, no guarantee can be given that the software is free of programming errors. For that reason, users are urged to find methods of cross-checking their results, even if in a partial way (verifying gauge invariance comes to mind). Please report *all* bugs, major and/or minor, should you find any. Simply send the author an e-mail to the address `paulo.nogueira@ist.utl.pt`.

That is not to say that each version of QGRAF doesn't have to go through many tests before being released. However, as the complexity of a computer program keeps growing it becomes impossible, at least in practical terms, to analyse all the possible cases that may be submitted to that program. With respect to the consistency checks that were performed, a special mention is due to FORM [3] and its convenient pattern matching capabilities.

Please do not distribute the code; instead, share the anonymous ftp site for the program, which is the following (as of May 2004).

```
ftp://cfif.ist.utl.pt:/pub/qgraf/
```

In this way the latest original version will be accessed.

The release consists of the FORTRAN source code (file `qgraf.f`), the user's guide (file `qgraf.ps`), and a few auxiliary files. As explained before, the files whose name ends in `.sty` are style files.

QGRAF has benefited from suggestions from several people, specially J. Vermaseren, T. van Ritbergen, and K. Chetyrkin. I also thank G. J. van Oldenborgh for sharing his experience on FORTRAN compilers.

References

- [1] P. Nogueira, J. Comput. Phys. 105 (1993) 279–289.
- [2] D.E. Knuth, The TeXBook (Addison Wesley, 1994).
- [3] J. Vermaseren, Symbolic Manipulation with FORM (Computer Algebra Netherlands, Amsterdam, 1991).
- [4] T. van Ritbergen, A.N. Schellekens, J.A.M. Vermaseren, Int. J. Mod. Phys. A 14 (1999) 41–96.