A Feynman graph generator for any order of coupling constants

Toshiaki KANEKO

Meiji-Gakuin University
Kamikurata-cho, Totsuka-ku, Yokohama, 244 Japan

abstract

A computer program is developed, which generates Feynman graphs automatically for scattering and decay processes in non-Abelian gauge theory of high energy physics. A new acceleration method is presented for generating and eliminating graphs. This method is shown to work quite efficiently for any order of coupling constants in any kind of theoretical model. A utility program is also available for drawing generated graphs. These programs consist of the most basic parts of GRACE system, which is now used to calculate tree and one-loop processes in an automatic way.

1-e-mail address: kane@minami.kek.jp
1 Introduction

In regard with the recent projects of experimental high energy physics with increasing available energy, accurate theoretical analysis has been required for increasing number of scattering processes. Since electro-weak theory combined with QCD, known as the standard model, is considered to be the basic theory among various theoretical models, exact perturbative calculation in the frame of this model is considered to be the standard of theoretical predictions. Because of the complexity of interactions, the number of Feynman graphs for one process becomes much greater than that in QED. There appear several tens to hundreds of graphs in tree process and several hundreds to thousands in one-loop process. In addition, in contrast to pure QCD, one cannot ignore various mass parameters which play an important role. These situations make the theoretical calculation of amplitude harder and tedious. At present it seems that the amount of labor for the exact calculations almost reaches at the limit of hand calculation. It is natural then to facilitate computers to carry out these works. Several groups have started independently to develop computer systems which automate the perturbative calculations in the standard model\[1, 2, 3, 4]\[5, 6\]. Automatic calculation of tree processes and some part of one-loop processes have been already achieved by these systems.

An automatic calculating system starts with generation of Feynman graphs for a given physical process. Feynman graph generation for all order in QED is not difficult, since the problem can be reduced to a problem equivalent to tree graph generation\[5, 6\]. For electro-weak theory or even in $\phi^3$ model, however, the situation is different. They include self-interactions of particles, whose vertices are symmetric under exchanging their legs. This symmetry of vertices results a complicated structure of internal symmetry to a Feynman graph after vertices are connected. It is not difficult to generate sufficient set of Feynman graphs by simply connecting vertices. The problem is, however, to eliminate duplicated graphs. It is hard to analyze the structure of internal symmetry of graphs so as to control graph generation process in avoiding duplicated graphs. Usually, a newly generated graph is discarded when it is found to be topologically equivalent to one of already generated. Explicit comparisons of graphs are of course possible. However, any known algorithm consumes time which increases as an exponential of the number of vertices. The graph comparison is generally not considered to all in to the category of polynomial time complexity\[7\]. Among known exponential time algorithms
orderly algorithm\cite{8} is suitable for graph generation. It is enough to look whether a newly generated graph is necessary or duplicated, without keeping other graphs. Based on this algorithm, a Feynman graph generator for wide variety of models was developed by P.Nogueira\cite{9}.

The aim of this paper is to present a computer program of Feynman graph generation fast enough for practical use. For this purpose, we have developed new method of graph generating which accelerates orderly algorithm. We classify vertices in such a way that there is no topologically equivalent vertex in different classes. With this classification, we are able to reduce the number of graphs to which orderly algorithm is applied and the number of vertices to be compared each other. The classification method of vertices, which we use, is an empirical one developed by graph theorists in order to determine the equivalence of given two graphs\cite{10}. We have modified this method to a suitable form for graph generation.

Our method of graph generation starts with isolated vertices and connect them one by one until vertices have no free legs, exhausting all possible combination of connection. In connecting vertices, we keep an intermediate graph whose vertices are partly connected. From a given graph, a number of different final graphs will be generated. So it is desirable to eliminate intermediate graphs in the stage as earlier as possible, if they are to be discarded. We have modified the classification method so as to be able to detect an intermediate graph being necessary or not\cite{11}. This method decreases the number of generated graphs which should be compared by orderly algorithm.

In order to make the problem simpler, we assume the following conditions:

1. Vacuum-to-vacuum graphs are not considered.

2. External particles of the graphs are assumed topologically different.

The first condition stems from the fact that the purpose of our automatic system is to calculate physical processes, namely scattering and decay of particles. With the second condition, we have one-to-one correspondence between Feynman graphs and Feynman amplitudes. One can generalize the method to loose these restrictions without essential difficulty, since orderly and classification algorithms are still applicable. However, it must make the program more complicate. This generalization will be discussed in section 5.

In the next section we introduce our basic methods and describe how to
accelerate orderly algorithm. Explanations of orderly and vertex classification algorithms are also given.

The method of graph generation is given in section 3. The vertex classification is modified and applied in a suitable way for intermediate elimination of graphs.

In section 4 we introduce the implementation of algorithms and utility programs.

Summary and discussions about physical processes calculated by GRACE system up to now as well as to generalize our program without two conditions mentioned above. extending our program in the last section.

Appendices A, B, C and D are brief manual for implementation of the programs and file formats.

2 Basic method

A graph consists of finite numbers of nodes (external particles and vertices) connected by edges (propagator or connection between an external particle and a vertex). Throughout this paper, we denote $N$ for the set of nodes, $E \subset \{(u,v) | u, v \in N\}$ for the set of edges. A graph $G = (N, E)$ is defined mathematically as the pair of the sets of nodes and edges. The definition of an edge includes its attributes such as the number of multiple connections between vertices, assigned name of particle of a propagator, etc. In order to make our discussion simpler, we drop these attributes in the following description of our method. A node has several legs which are connected to the end points of edges. An external particle has one leg and a vertex has a definite number of legs corresponding to the number of interacting particles. A node is uniquely labeled by an integer number. We identify a node by its labeling number. So we regard the set of nodes $N$ being the same set of integer numbers $\{1, ..., |N|\}$.

The graph generation process connects two legs of vertices in forming an edge one by one, starting from isolated nodes, until all legs of nodes are connected. A simple method of graph generation will produce isomorphic graphs, which are only different in the way of labeling vertices. Our problem is how to pick up a representative systematically from each isomorphic set of graphs. In this section, we consider how to eliminate graphs when all possible graphs are generated. It is required to discard irrelevant graphs not only at
the final form but also incomplete intermediate graphs, in order to reduce the total number of graphs to be compared. Discussion on the problem of earlier elimination is given in the next section.

**Orderly algorithm** compares graphs through a function, called **coding**, which maps a graph to an integer number so that different graphs in a set of isomorphic graphs are mapped to different values. Such a function is easily realized. For example, it is sufficient to regard the elements of adjacency matrix as a sequence of digits of an integer number.

Let us take a graph $G$ and consider the permutation group $P$ acting on the set of nodes $N$ of the graph. A permutation $p \in P$ can be generalized to act on the set of edges by defining $p(u,v) = (pu, pv)$ for $(u,v) \in E$, $u, v \in N$ and generalized to act on the graph $G$ as $pG = p(N, E) = (pN, pE) = (N, pE)$. Two graphs $G = (N, E)$ and $G' = (N', E')$ are isomorphic when there exists such a permutation $p$ that maps the graph $G$ to $G' = pG$. This condition is equivalent to:

$$(u,v) \in E \iff (pu,pv) \in E'. \quad (1)$$

The set $\{pG | p \in P\}$ is the collection of all generated isomorphic graphs of $G$. In order to pick up the representative from this set, orderly algorithm selects the graph by the following condition with coding $f$:

$$f(G) = \max_{H \in \{pG | p \in P\}} f(H) = \max_{p \in P} f(pG). \quad (2)$$

A graph $G$ is indirectly compared with graph $pG$ through their value of the coding.

A permutation $p \in P$ is an automorphism of a graph $G$ when $pG = G$. The set of automorphism of a graph $G$ forms a group $\Gamma_G$ called **automorphism group**. If a permutation $p$ satisfies the condition (2), $p$ is found to be an element of $\Gamma_G$. In this way, one can construct $\Gamma_G$ explicitly. The symmetric factor $|\Gamma_G|$ necessary for calculating Feynman amplitude is evaluated in the same time by the orderly algorithm.

What one must do with orderly algorithm to select graphs is:

1. Generate all possible graphs.
2. Apply all possible permutations to all generated graphs.
3. Discard a graph whose value of the coding is less than one of a permuted graph.

A simple application of this algorithm requires to test $O(|N|!)$ permutations for $O(|N|!) = |N|!$ isomorphic graphs:

$$f(G) = \max_{H \in \{pG|p \in Q\}} f(H) = \max_{p \in Q} f(pG).$$

(3)

One should notice that such a subgroup $Q$ is not necessary to be identical among all graphs, but is sufficient to be unique among isomorphic graphs. Replacement of $\{pG|p \in P\}$ by $\{pG|p \in Q\}$ in the orderly algorithm requires that generated set of isomorphic graphs of $G$ should be identical to the set $\{pG|p \in Q\}$. We see below this condition is satisfied by taking the following subgroup $Q$ combining with an appropriate graph selection condition.

In order to realize such a subgroup $Q$, we consider a classification $\{N_i\}$, $\bigcup_i N_i = N$ of the set of nodes $N$ such that any two vertices in different classes are topologically different. In other words, each class of vertices is a direct sum of the orbits in $\Gamma_G$ (the orbit of node $v$ in $\Gamma_G$ is the set $\{u|\exists p \in \Gamma_G, p^v = u\}$). We call such a classification consistent classification. With this classification of vertices, we can take a group $S_{|N_i|} \otimes S_{|N_i|} \otimes \ldots$ for the subgroup $Q$, where $S_k$ is the symmetry group acting on a set of $k$ elements. The number of the permutations to be tested is reduced from $|N|!$ to $|N_1|! \times |N_2|! \times \ldots$. The outline of our vertex classification method is as follows; first nodes are classified by their number of legs. Since we consider external particles are topologically different each other, each of them forms its own class. We call this initial classification as primitive classification. Starting from the primitive classification, the classes are refined repeatedly by the following algorithm. We consider a vector $a_v[i], i = 0, 1, \ldots, |N|$ for a node $v \in N$. The zeroth element of the vector $a_v[0]$ keeps its current class number. The $i$-th element of the vector keeps the number of nodes in the class $N_i$ which are connected to the node $v$:

$$a_v[0] = j, \quad \text{for } v \in N_j,$$

$$a_v[i] = |\{(u, v) \in E|u \in N_i\}| \quad \text{for } i > 0.$$

(4)

Vertices are classified again by the value of these vectors. New classes are numbered in the increasing order of the vectors by the lexicographical ordering of its vector $a_v$. In this way, the nodes are classified only by their
topology. It is obvious that the automorphism group $\Gamma_G$ is a subgroup of $Q$ and our classification is consistent. We summarize this refining method of classification as follows;

void refine\{\{N_i\}\}
{
    do 
    for all (i) 
    
        $M_i = N_i$;
        for all ($v \in N_i$)
        
            $a_v[0] = i$;
            
    for all ($v \in N$)
    for all (i)
    
        $a_v[i] = |\{(v, w) \in E | w \in N_i\}|$;
        (update \{N_i\} such that
        
            $a_v = a_w, v \in N_i, w \in N_j \Rightarrow i = j$;
            $a_v < a_w, v \in N_i, w \in N_j \Rightarrow i < j$;
    
    )
    } while ( \{N_i\} != \{M_i\} )
}

We note that the input classes of this algorithm is not necessary to be primitive classification. This property is suitable to use the algorithm iteratively in the graph generation process, as described in the next section. Since output classes are refined ones of input classes, there exists such an input class $N^\text{in}_k$ that $N^\text{out}_i \subset N^\text{in}_k$ for any $N^\text{out}_i$. Moreover, the following ordering condition on the class numbering holds:

$$\forall i \forall j \left[ N^\text{out}_i \subset N^\text{in}_k, N^\text{out}_j \subset N^\text{in}_l, i < j \Rightarrow k \leq l \right].$$

This condition is confirmed by the fact that the algorithm renumbers the new classes in the ordering of vector $a_v$, which keeps the old class number at the first element.

In order to limit the set of generated graphs which is consistent with this subgroup, we impose the following ordering condition for the generated graphs:

$$\forall i \forall j \forall u \in N_i \forall v \in N_j \left[ i < j \Rightarrow u < v \right].$$
where ordering of vertices is evaluated in terms of labeling numbers of vertices. We number the nodes at the beginning so that the primitive classification satisfies this condition. The relation (5) shows that a graph satisfying condition (6) for \( \mathcal{N}_i^{\text{out}} \) is found in the set of graphs which satisfy the same condition for \( \mathcal{N}_i^{\text{in}} \).

It is easy to see sufficient graphs are surviving after selecting by this condition. We consider a graph \( G \) which satisfies this condition for \( \mathcal{N}_i^{\text{in}} \), but not for \( \mathcal{N}_i^{\text{out}} \). It is possible to make the graph to satisfy the condition for \( \mathcal{N}_i^{\text{out}} \) by properly renumbering the nodes within each \( \mathcal{N}_i^{\text{in}} \) with a permutation \( p \). The graph \( pG \) should have been generated, since graph generation method have exhausted all possibility of connecting vertices. Thus, for any graph \( G \), there exists a graph which is isomorphic to \( G \) and selected by the condition for \( \mathcal{N}_i^{\text{out}} \).

By our classification method, we assure the fact that the class number is uniquely defined among isomorphic graphs, since the classes are constructed and numbered only from the topological structure of the graph. That is, when vertices of two isomorphic graphs \( G \) and \( H = pG \) are classified to \( \mathcal{N}_i \) and \( \mathcal{M}_j \), respectively, any class \( N_i \) of \( G \) is mapped to the class \( M_i = pN_i \) with the same index by the isomorphism \( p \). Especially, \( |N_i| = |M_i| \) is satisfied for all \( i \). The selection condition (6) can be rewritten as:

\[
\forall u \in N_i \left[ \sum_{j<i} |N_j| < u \leq \sum_{k\leq i} |N_k| \right].
\]  

(7)

Since this is also true for \( \{M_i\} \), the corresponding classes between \( G \) and \( H \) are identical \((N_i = M_i, \text{ for all } i)\). The isomorphism \( p \) maps each \( N_i \) to itself \( pN_i = N_i \) and then the isomorphism \( p \) is shown to belong to \( Q \).

We have confirmed first that necessary graphs are kept after selection by condition (6). For the second, any two surviving isomorphic graphs are transformed each other by a element of the subgroup \( Q \). Now the orderly algorithm (3) can be properly applied to the generated and then selected graphs.
3 Graph generation and selection

The nodes are connected iteratively starting from one fixed external particle, which we call the root of the graph. We define level of a node by counting the distance from the root. The level of a node is infinite when the node is not in the connected component of the root. In the connecting process, we take the source node of connection in the increasing order of the value of the level. The root, with one leg, is first connected to another node. The connected node becomes the node at the first level. Then the node is connected to other nodes until all of its legs are connected. These connections define the set of nodes in the second level. Then the all legs of all nodes in the second level are connected to other ones. This process is proceeded in a recursive way so as to exhaust all possibilities of selecting target node of the connections. The skeleton description of the algorithms is described as follows:

```c
void gsconn()
{
    node ns, nt;

    ns = (find node at the lowest level with free legs);
    if (no more such nodes) {
        if (the graph is connected one) {
            if (the graph is accepted by orderly algorithm)
                (a new graph is obtained);
        }
    } else {
        for all (node nt with free legs) {
            (connect ns to nt);
            gsconn();
            (disconnect ns from nt);
        }
    }
}
```

For efficient graph generation, it is important to judge whether acceptable graphs can be produced from the current intermediate graph or not. We apply the condition (6) after classifying vertices not only to the final form
of the generated graphs but also to the intermediate graphs which appear in
the way of graph generation.

One must note that the classification of an arbitrary intermediate graphs
is not always consistent with that of the final form of the graphs. For example,
we consider a final form of a graph Fig.1a and its intermediate graph Fig.1b.
The set of classes of nodes of Fig.1a is

$$\{ (1), (2), (3), (4, 5), (6) \} ,$$

while one of Fig.1b is

$$\{ (1), (2), (3), (4), (5), (6) \} .$$

This example shows that vertices 4 and 5 are equivalent in the final form
of the graph, although they are not always equivalent in the way of graph
generation.

Figure 1: An example of two loop graph.

We can recover the consistent classification by considering an intermediate
graph $G^0 = (N, E^0)$ at the time just when all legs of all vertices in the level
$l$ are connected. These intermediate graphs forms a finite series:

$$G^0 \subset G^1 \subset \ldots \subset G^{l-1} = G,$$
where $m$ is the maximum level of vertices in the graph $G$. Fig.1c represents an example of $G^{[l]}$. The set of edges $E^{[l]}$ is identical to the set of edges:

$$E^{[l]} = \{(u,v) \in E | \min(\text{level}(u), \text{level}(v)) \leq l\}. \quad (8)$$

The following property holds for the automorphism group $\Gamma_{G^{[l]}}$ of $G^{[l]}$:

$$\Gamma_{G^{[l]+1}} \subset \Gamma_{G^{[l]}}. \quad (9)$$

If this is violated, there exists a permutation $p \in \Gamma_{G^{[l]+1}}$ which is not an element of $\Gamma_{G^{[l]}}$. This means that there exists an edge $(u, v) \in E^{[l]}$ and $(pu, pv) \notin E^{[l]}$. On the other hand, since $p$ is an automorphism in $\Gamma_{G^{[l]+1}}$ and $\text{level}(pu) = \text{level}(u)$, the following equation holds:

$$\min(\text{level}(pu), \text{level}(pv)) = \min(\text{level}(u), \text{level}(v)) \leq l, \quad (10)$$

which implies $(pu, pv) \in E^{[l]}$.

The relation (9) implies $\Gamma_{G} = \Gamma_{G^{[l]+1}} \subset \Gamma_{G^{[l]}}$, and then an orbit of $\Gamma_{G^{[l]}}$ is a direct sum of orbits of $\Gamma_{G}$. Since a class constructed by our method is a direct sum of orbits of automorphism group of the graph, a class defined for $G^{[l]}$ is a direct sum of orbits of $\Gamma_{G^{[l]}}$, and then is that of $\Gamma_{G}$. So the vertex classification in $G^{[l]}$ is consistent with the classification in $G$.

Now we can eliminate irrelevant intermediate graphs by the condition (6). Further we can use the classes for $G^{[l-1]}$ as the input for constructing $G^{[l]}$. The final form of our algorithms is as follows:

```c
void gsconn()
{
    node ns, nt;

    if all (legs of the vertices in the current level have been connected) {
        (refine classes of the vertices);
        if(! (class ordering condition (6) is satisfied))
            return;
    }

    ns = (find node at the lowest level with free legs);
    if(no more connectable legs) {
        if (the graph is connected one) {
            if (the graph is accepted by orderly algorithm)
```
(a new graph is obtained);  
}  
} else {
    for all (node nt with free legs) {
        (connect ns to nt);
        gsconn();
        (disconnect ns from nt);
    }
}

4 Implementation

We have implemented the method described above as a computer program written in C language. The program first generates topology and after that particles are assigned to the propagators. The particles of the propagators are determined in accordance with the table of particles and vertices defined by users. It is possible that duplicated graphs are produced in the way of particle assignment. They are eliminated again by orderly algorithm with extended coding for graphs with particle attributes on the edges.

The program has the following options for graph generation:

1. to pick up only one-particle irreducible graphs[12].
2. not to generate graphs with self-energy part at an external particle line.
3. not to generate graphs with self-energy part at an internal particle line.
4. to generate skeleton graph, where looped one-particle irreducible subgraphs are considered as blobs. The generated graphs are tree but with blob vertices.
5. not to assign particles to the internal lines (only topology).
6. to generate graphs with counter terms for renormalizing the theory.

Counter terms are automatically generated when their interaction are same form as tree vertices. Other counter terms can be added by user.
The way of specifying these options are described in the appendix B.

We also provide graph drawing facility on X-window system and on PostScript files. We show an example of output figures in Fig. 2.

Figure 2: An example of drawn graphs.

The total numbers of graphs depends on the conditions of graph selection. In special cases, they are analytically calculable. In $\phi^4$ theory, the method of enumerating connected graphs is developed in group theoretical method [13, 14]. When there appear different kinds of particles in the model, weighted sum by the symmetric factor can be calculated in zero dimensional field theory [15]. These numbers of graphs are calculated and listed in ref. [9]. We have checked our program by comparing with these numbers. Since up to 5 loop graphs, orderly algorithm is not necessary to generate unique graphs as discussed in ref [11], we have checked also the number of graphs for 6 loop tadpole which includes 90156 graphs. The number of one particle irreducible $\phi^4$ graphs are checked by counting in recursion formula for one- and two-loop graphs.
<table>
<thead>
<tr>
<th>E</th>
<th>L</th>
<th>count</th>
<th>HP9000</th>
<th>SunIPX</th>
<th>DEC5000</th>
<th>Gateway</th>
<th>PC98VX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>46</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>23.00</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>471</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>15.19</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>15861</td>
<td>880.50</td>
<td>231.77</td>
<td>164.21</td>
<td>335.26</td>
<td>10.51</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>90156</td>
<td>550.54</td>
<td>141.32</td>
<td>90.75</td>
<td>210.32</td>
<td>6.49</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>351</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>18.47</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5076</td>
<td>1033.81</td>
<td>281.53</td>
<td>211.50</td>
<td>416.75</td>
<td>12.53</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>84749</td>
<td>719.49</td>
<td>194.37</td>
<td>134.03</td>
<td>289.20</td>
<td>8.78</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>14</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>217</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>19.73</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3729</td>
<td>1154.49</td>
<td>313.36</td>
<td>236.01</td>
<td>465.54</td>
<td>13.81</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>70600</td>
<td>845.00</td>
<td>227.52</td>
<td>162.90</td>
<td>335.79</td>
<td>10.13</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>99</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>19.80</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2214</td>
<td>—</td>
<td>339.05</td>
<td>273.33</td>
<td>506.64</td>
<td>14.96</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>50051</td>
<td>933.79</td>
<td>255.22</td>
<td>191.25</td>
<td>377.89</td>
<td>11.31</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>25</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>947</td>
<td>—</td>
<td>361.45</td>
<td>286.97</td>
<td>529.05</td>
<td>15.52</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>28365</td>
<td>997.01</td>
<td>275.12</td>
<td>211.05</td>
<td>399.23</td>
<td>12.16</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>220</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>18.33</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>11460</td>
<td>1049.45</td>
<td>291.75</td>
<td>223.39</td>
<td>417.18</td>
<td>12.82</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>2485</td>
<td>1230.20</td>
<td>353.49</td>
<td>276.11</td>
<td>514.49</td>
<td>14.79</td>
</tr>
</tbody>
</table>

| E   | : The number of external particles. |
| L   | : The number of loops. |
| count | : The number of connected graphs. |
| HP9000 | : HP9000/755 |
| SunIPX | : Sun IPX |
| DEC5000 | : DECStation 5000 |
| Gateway | : Gateway2000, i486DX2, 66MHz, Linux + gcc |
| PC98VX | : PC98VX21, i286, 10MHz, MS-DOS + Turbo C |
| — | : Execution time is less than two seconds. |

Table 1: The number of generated graphs per second in $\phi^4$ theory.
The performance of the program is shown in table 1 in the case of connected graphs in $\phi^4$ model. The theoretical prediction of number of graphs is calculated by a REDUCE program. Since it takes much cpu-time than our graph generation program, we limit the number of loops as the number of external legs increases. We have measured time consumed with "-a -c" options described in appendix A, for the processes for which the number of graphs counted by the REDUCE program.

5 Summary and discussions

We have developed a program which generates Feynman graphs on the accelerated method of vertex classification and orderly algorithm. The order of coupling constants of a physical process is not limited. Graphs can be generated based on a user defined model, in this paper we adopt the usual standard model as an example. The program has several options of graph selection including to generate renormalization counter terms. Although the asymptotic behavior of the execution time is proportional to the factorial of the number of nodes, the program is fast enough for practical use.

This program is a part of GRACE system, which automates tree and one-loop scattering processes. With this system, cross-section including one-loop corrections in the processes $e^+e^- \rightarrow HZ, e^+e^- \rightarrow t\bar{t}$ and one-loop gluon corrections to the process $e^+e^- \rightarrow q\bar{q}\gamma$ are automatically calculated[2]. For tree case, cross-section of scattering processes with up to five final particles are calculated as shown in table 2. The number of graphs are counted in covariant and Feynman gauge for tree and one-loop process, respectively. In both cases, interactions between Higgs particle and light fermion are neglected. Integration over phase space is calculated by integration package BASES[16].

Our program is available through anonymous ftp from the directory kek/minami/grc on the workstation ftp.kek.jp.
<table>
<thead>
<tr>
<th>Process</th>
<th>#Graph</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^+ e^-$ → $W^+ W^- \gamma$</td>
<td>18</td>
<td>[17a]</td>
</tr>
<tr>
<td>→ $\nu_e \bar{\nu}_e Z^0$</td>
<td>9</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $\nu_e \bar{\nu}_e H$</td>
<td>2</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $e^+ e^- H$</td>
<td>2</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $Z^0 Z^0 H$</td>
<td>4</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $W^+ W^- H$</td>
<td>11</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $Z^0 Z^0 Z^0$</td>
<td>9</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $W^+ W^- Z^0$</td>
<td>20</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $t \bar{t} Z^0$</td>
<td>9</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $t \bar{t} H$</td>
<td>6</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $H H Z^0$</td>
<td>6</td>
<td>[17b]</td>
</tr>
<tr>
<td>→ $\gamma \gamma \gamma$</td>
<td>9*</td>
<td>—</td>
</tr>
<tr>
<td>$\gamma \gamma$ → $e^+ e^- Z^0$</td>
<td>6</td>
<td>—</td>
</tr>
<tr>
<td>$\gamma e$ → $e W^+ W^-$</td>
<td>18</td>
<td>—</td>
</tr>
<tr>
<td>$gg$ → $q \bar{q} \gamma$</td>
<td>18</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2a Tree processes with final 3-body

<table>
<thead>
<tr>
<th>Process</th>
<th>#Graph</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^+ e^-$ → $\nu_e \bar{\nu}_e W^+ W^-$</td>
<td>60</td>
<td>[17c]</td>
</tr>
<tr>
<td>→ $\nu_{\mu} \bar{\nu}_{\mu} W^+ W^-$</td>
<td>36</td>
<td>[17c]</td>
</tr>
<tr>
<td>→ $e^+ e^- W^+ W^-$</td>
<td>114</td>
<td>[17c]</td>
</tr>
<tr>
<td>→ $\nu_e \bar{\nu}_e Z^0 Z^0$</td>
<td>57</td>
<td>[17c]</td>
</tr>
<tr>
<td>→ $e^+ \nu_e W^- Z^0$</td>
<td>88</td>
<td>[17c]</td>
</tr>
<tr>
<td>→ $e^+ e^- Z^0 Z^0$</td>
<td>86</td>
<td>[17c]</td>
</tr>
<tr>
<td>→ $e^+ e^- \gamma \gamma$</td>
<td>80</td>
<td>[17d]</td>
</tr>
<tr>
<td>→ $t \bar{t} W^+ W^-$</td>
<td>40</td>
<td>[17d]</td>
</tr>
<tr>
<td>→ $\nu_e \bar{\nu}_e b \bar{b}$</td>
<td>21</td>
<td>[17e]</td>
</tr>
<tr>
<td>→ $\nu_{\mu} \bar{\nu}_{\mu} b \bar{b}$</td>
<td>11</td>
<td>[17e]</td>
</tr>
<tr>
<td>→ $e^+ \nu_e \bar{b} \bar{b}$</td>
<td>21</td>
<td>[17f]</td>
</tr>
<tr>
<td>→ $W^+ W^- \gamma \gamma$</td>
<td>138</td>
<td>—</td>
</tr>
<tr>
<td>→ $c^- \nu_e u \bar{d}$</td>
<td>24</td>
<td>[17g]</td>
</tr>
<tr>
<td>→ $u d \bar{u} \bar{d}$</td>
<td>69</td>
<td>[17g]</td>
</tr>
<tr>
<td>→ $q \bar{q} \gamma$</td>
<td>21</td>
<td>—</td>
</tr>
<tr>
<td>→ $e^+ e^- b \bar{b}$</td>
<td>50</td>
<td>—</td>
</tr>
<tr>
<td>$\gamma e$ → $\nu_e W^- H H$</td>
<td>40</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2b Tree processes with final 4-body
<table>
<thead>
<tr>
<th>Process</th>
<th>#Graph</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^+e^- \rightarrow \mu^+\nu_\mu b\bar{b}\gamma$</td>
<td>71</td>
<td>—</td>
</tr>
<tr>
<td>$e^- \bar{b}\bar{u}d\gamma$</td>
<td>142</td>
<td>[17h]</td>
</tr>
</tbody>
</table>

Table 2c: Tree processes with final 5-body

<table>
<thead>
<tr>
<th>Process</th>
<th>#Graph</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^+e^- \rightarrow HZ$</td>
<td>89</td>
<td>[2]</td>
</tr>
<tr>
<td>$t\bar{t}$</td>
<td>50</td>
<td>[2]</td>
</tr>
</tbody>
</table>

Table 2d: One-loop processes with final 2-body

<table>
<thead>
<tr>
<th>Process</th>
<th>#Graph</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^+e^- \rightarrow q\bar{q}\gamma$</td>
<td>12</td>
<td>[2]</td>
</tr>
</tbody>
</table>

Gluon correction with only final radiations.

Table 2e: One-loop processes with final 3-body

Table 2: Scattering processes calculated by GRACE system.
In the following we discuss how to loose the restrictions for graph generation assumed in introduction. The second restriction about identical external particle can be removed by changing primitive classification so as to put identical particles in the same class. At the same time, root node should be replaced by a class of nodes.

The first restriction of excluding vacuum-to-vacuum graphs is more complicated to loose. It is possible that the primitive classification contains only one class, such as in vacuum-to-vacuum graphs in $\phi^3$ theory. In this case, vertex classification and acceleration methods do not work anymore. However, one can artificially fix one vertex as the root of the graph.

We consider a set of graphs $A$ generated by our method with fixing the vertex 1 among $n = |N|$ vertices and selected by ordering condition (6). Let us also consider the true set of vacuum-to-vacuum graphs $T$. For any graph $G \in T$, one can construct the set of isomorphic graphs $T_G = \{ pG \ | \ p \in S_{n-1} \}$, where $S_{n-1}$ is the symmetric group acting on the set of vertices $\{2, \cdots, n\}$. When one selects the graphs from $T_G$ by applying ordering condition (6) after classifying vertices with fixing vertex 1, all the selected graphs are belong to $A$, since $A$ is constructed by the same selection rule applied to all possible graphs. So the set of graphs $A$ contains at least an isomorphic graph to any graph of $T$, but some graphs in $A$ are duplicated.

Duplication of graphs is tested by orderly algorithm (2) in terms of the permutation group $P = S_n$ of the vertices. Since we have already eliminated some of duplicated graphs by the ordering condition (6), one must limit the set of graphs $\{ pG \ | \ G \in A, \ p \in P \}$ by the same condition before comparing the values of the coding. The orderly algorithm is then changed to the following form:

```plaintext
accept = True;
for all (H = pG, p ∈ P) {
    (classify vertex of H by fixing vertex 1);
    if(ordering condition (6) is satisfied) {
        if(coding(G) < coding(H)) {
            accept = False;
            break;
        }
    }
}
if(accept)
```

18
(accept graph $G$);

6 Acknowledgement

The author wishes to thank members of Minami-Tateya collaboration for continuous discussions and many kind of supports. Especially, he owes special thanks to Dr. Y. Shimizu, whose influence appears in the basis of this work, for his useful suggestions in reading the manuscript and to Dr. S. Kawabata who provided the first version of the driver program of X-Library.

The author is also indebted for useful discussions to Dr. P. Nogueira about orderly algorithm and to Dr. J. A. M. Vermaseren on the format of model definition and output files.

This work is supported in part by Ministry of Education, Science and Culture, Japan under Grant-in-Aid for International Scientific Research Program No.04044158.
References


H.Eck and J.Küblbeck, ibid., pp.565.


[7] For example, C.M.Hoffman, Lecture Notes in Computer Science 136 Springer-Verlag, 1982;


Univ. Técnicos de Lisboa preprint, IFM-10/93, 1993


[12] Method of detecting one-particle irreducibility is almost same as one of decomposing a graph to blocks:
J.Hopcroft and R.Tarjan, Commun. of ACM, 16 (1973) 372;


For new version, S.Kawabata, to be pubushed.

h: J.Fujimoto et al., in Proc. of Phys. at LEP200 and beyond.
A Installation and execution

The source code is available by anonymous ftp from ftp.kek.jp in the directory kek/minami/grc. It has two programs, one generates Feynman graphs and the other draws generated graphs which work on UNIX with X-Window system.

The graph generation program reads two files, a process file and a model file. It creates output file named “out.grf”, which keeps the information about the generated graphs. The file format of input files are described in appendices B and C.

The graph drawing program reads the model and the output file of generation program and draws the graphs on the display with X-Window system. This program is not so intelligent to display graphic objects beautifully, however, it will still useful to check the generated graphs. This program has several subcommands.

The procedure of installation is:

1. Uncompress and expand the source code file.
   In a new directory, run the following command.
   
   \[ zcat grc.tar.Z | tar xvf - \]

2. Compilation
   In the subdirectory src, run make command. Before it, check the path name of Xlib in the Makefile. If athena widget (included in the standard distribution of X-Window system) is installed in your system, use the line in the Makefile:
   
   \[ XCOPT = -DTOOLKIT -DX/1/1 \]
   if not so, use:

   \[ XCOPT = -DX/1/1 \]

3. Test Feynman graph generator
   In the subdirectory sample, run the following command

   \[ ./src/grc \]
Then the program generates an example of tree graphs for $e^+e^- \rightarrow W^+W^-$ process, 316 one-loop graphs of the same process and 28 tree graphs for $e^+e^- \rightarrow W^+W^-\gamma$.

4. Test Feynman graph drawer
   In the same subdirectory sample, run the following command
   
   ```
   ../src/grcdraw -h
   ../src/grcdraw
   ```
   
   Try to type “f”, “b”, “n”, “p”, “g” “q”.
   If you have installed with athena widget, buttons will be appeared on the display.

**Feynman graph generator**

The command syntax of `grc` command is:

```
grc [options] [process-file-name]
```

The `process-file-name` is an input file specifying the physical process and options, which format is described in appendix B. The default process file name is “in.prc”.

The command line options of the `grc` command is

- `-a`
  
  Skip particle assignment.

- `-c`
  
  Not generate output file but only count the number of graphs.

- `-o output-file-name`
  
  Specify output file name. The default output file is “out.grf”

- `-h`
  
  Print help message.

Try also with the help option:

```
../src/grc -h
```
Feynman graph drawer

The command syntax of \texttt{grcdraw} command is:

\begin{verbatim}
grcdraw [options] [graph-file-name]
\end{verbatim}

The \textit{graph-file-name} is an output file generated by \texttt{grc} command, which format is described in appendix D. The default process file name is "\texttt{out.grf}".

The command line options of the \texttt{grcdraw} command is:

- \texttt{-w number-of-graphs}
  If this option is specified \((\text{number-of-graphs})^2\) graphs appears on the display.

- \texttt{-h}
  Print help message.

Try also with the help option:

\begin{verbatim}
../src/grcdraw -h
\end{verbatim}

This program has some commands for selecting graphs drawn on the display. They are as follows:

\begin{verbatim}
qu : quit.
n : display next process
p : display previous process
f : display forward graphs
b : display backward graphs
g : scale up the size of graphs
s : scale down the size of graphs
l : display particle names of internal lines (on/off)
t : display particle names and graph number (on/off)
<number>j : jump to the specified graph
o : output the displayed graphs to PostScript file
\end{verbatim}

When the program is installed with athena widget, such commands are also displayed as buttons.
B Process file

The graph generation program works in accordance with the process file. The following is an example of the file for $e^+e^- \rightarrow W^+W^-\gamma$ process of tree and one-loop.

```plaintext
Model="allmdl";

Process;
ELWK={2, 4};
Initial={electron, positron};
Final ={W-plus, W-minus};
Expand=Yes;
OPI=No;
Pend;
Process;
ELWK=3;
Initial={electron, positron};
Final ={W-plus, W-minus, photon};
Expand=Yes;
OPI=No;
Pend;
```

The lines beginning with “%” are ignored as comment lines.

The first non-comment line specifies the file which describes the model used for the graph generation, described in appendix C.

Then follows the descriptions of processes. A block describing a process begins with “Process;” line and ends with “Pend;” line. In the block, descriptions are given for the order of coupling constants, initial particles, final particles and some other options.

The name of coupling constant (“ELWK” in the above example) is defined in the model file. When the value of coupling constant is given as a list of numbers, the program generates graphs for each value of coupling constants with same external particles.

Initial and final external particles are given as lists of particles names. They are defined in the model file.

Available options are:

- OPI = Yes | No
  Generate one particle irreducible graphs or not.
• **Expand** = Yes | No
  Expand looped part or generate looped parts as blobs.

• **Tadpole** = Yes | No
  Generate tadpoles or not.

• **extself** = Yes | No
  Generate self-energy part at external particle or not.

• **selfe** = Yes | No
  Generate whether with self-energy part or not at internal particle line.

• **countert** = Yes | No
  Generate renormalization counter terms or not.
C Model file

Here we describe the format of model file. Our definition of particles and vertices include not only necessary information for graph generation but also for amplitude generation. We show an example of this file:

```plaintext
Order={ELWK, QCD};
%----------------------------------------------
% gauge bosons
%----------------------------------------------
Particle=W-plus["W+"]; Antiparticle=W-minus["W-"];
  PType=Vector; Charge=1; Color=1; Mass=AMW; Width=AGW;
  MValue="80.220D0"; WValue="2.120D0"; PCode=2;
Pend;
...

%----------------------------------------------
% scalars
%----------------------------------------------
Particle=Higgs["H"];
   Antiparticle=Particle;
  PType=Scalar; Charge=0; Color=1; Mass=AMH; Width=AGH;
  MValue="150.000D0"; WValue="0.000D0"; PCode=31;
Pend;
%
...

%----------------------------------------------
% leptons
%----------------------------------------------
Particle=nu-e["nu-e"];
   Antiparticle=nu-e-bar["nu-e-"];
  PType=Fermion; Charge=0; Color=1; Mass=AMNE; Width=0;
  MValue="0.000D0"; PCode=51; Massless;
Pend;
...

Particle=electron["e-"]; Antiparticle=positron["e+"];
  PType=Fermion; Charge=-1; Color=1; Mass=AMEL; Width=0;
  MValue="0.511D-3"; PCode=55;
Pend;
...
```

27
%--------------------------------------------
% quarks
%--------------------------------------------
Particle=u; Antiparticle=u-bar[U]
  PType=Fermion; Charge=2/3; Color=3; Mass=AMUQ; Width=0;
  MValue="100.0D-3"; PCode=61;


%--------------------------------------------
% VVV
%--------------------------------------------
Vertex={Z, W-minus, W-plus}; ELWK=1; FName=ZWW;
  FValue={R, " GG*GCOS"}; Vend;
Vertex={W-minus, W-plus, photon, photon }; ELWK=1; FName=AWW;
  FValue={R, " GE"}; Vend;
Vertex={gluon, gluon, gluon }; QCD=1; FName=GGG;
  FValue={R, " CQCD"}; Vend;
%--------------------------------------------
% VVV
%--------------------------------------------
Vertex={W-plus, W-minus, photon, photon }; ELWK=2;
  FName=WWA; FValue={R, " GE2"}; Vend;
Vertex={W-plus, W-minus, Z, photon }; ELWK=2;
  FName=WWZA; FValue={R, " GE*GG*GCOS"}; Vend;

%--------------------------------------------
% FFV (FFW : without quark mixing)
%--------------------------------------------
Vertex={positron, nu-e, W-minus}; ELWK=1; FName=WNE;
  FValue={R, " GWFL"}; FType="V-A"; Vend;
Vertex={anti-muon, nu-mu, W-minus}; ELWK=1; FName=WMN;
  FValue={R, " GWFL"}; FType="V-A"; Vend;
Vertex={anti-tau, nu-tau, W-minus}; ELWK=1; FName=WNT;
  FValue={R, " GWFL"}; FType="V-A"; Vend;
Vertex={nu-e-bar, electron, W-plus }; ELWK=1; FName=WEL;
  FValue={R, " GWFL"}; FType="V-A"; Vend;
Vertex={nu-mu-bar, muon, W-plus }; ELWK=1; FName=WMU;
  FValue={R, " GWFL"}; FType="V-A"; Vend;
Vertex={nu-tau-bar, tau, W-plus }; ELWK=1; FName=WTA;
Counter terms introduced by photon-Z mixing.

SSV

Vertex=(chi-3, Higgs, photon ); ELWK=3; FName=AHY; 
Vend;

SVV

Vertex=(Higgs, Z, photon ); ELWK=3; FName=HZA;
Vend;
Vertex=(Higgs, photon, photon ); ELWK=3; FName=HAA;
Vend;

FFV (FFZ)

Vertex=(nu-e-bar, nu-e, photon ); ELWK=3; FName=ANE;
Vend;
Vertex=(nu-mu-bar, nu-mu, photon ); ELWK=3; FName=ANM;
Vend;
Vertex=(nu-tau-bar, nu-tau, photon ); ELWK=3; FName=ANT;
Vend;

Mend;

Lines beginning with “%” are comment lines.

The file is composed of three parts; definitions of the names of coupling constants, definitions of particles and then definitions of vertices and counter-terms, arranged in this order. Definition is an sequence of “keyword=value;” or “keyword;”. The value part may have an optional part, which is enclosed by brackets “[” and “]”.

The first keyword is Order, which define the name of coupling constants. In the case of multiple coupling constants appear in the model, the list of these names are specified enclosed in braces “{” and “}”. For example,
\texttt{Order=\{QED, QCD\}};

### C.1 Definition of particles

A definition of a particle begins with keyword \texttt{Particle} and ends with keyword \texttt{Pend}.

The value of the keyword \texttt{Particle} is the name of the particle. The name, beginning with an alphabet, should be unique among particles defined in this file, since this name is used to identify the particle. Option to the name is a shorthand name of the particle, which is used to show the particle in the graphic output.

In the same way, the name of anti-particle is given as the value of keyword \texttt{Antiparticle}. When the anti-particle coincides to the particle, this part should be defined as

\texttt{Antiparticle=Particle}.

The type of particle is given by keyword \texttt{PType}. Its value is either \texttt{Scalar}, \texttt{Vector}, \texttt{Majorana}, \texttt{Fermion} or \texttt{Ghost}.

The keyword \texttt{Charge} specifies the electric charge. The value is given as an signed integer or rational number in the unit of positron charge $e$.

The dimension of color representation of the particle is specified by \texttt{Color} keyword as an integer number.

The definition of the mass parameter of the particle is composed of three items, that is, Fortran variable name of the mass, default numerical value and flag specifying massive or Massless. Even if the particle is Massless, fictitious mass can be introduced in some part of calculation in order to avoid infrared divergence or mass singularity. We require a definition of particle being massless or massive, and the Fortran variable name for particle mass with default value not only for massive particle but also massless one.

The keyword \texttt{Massless} or \texttt{Massive} (without a value) specifies the particle is either massless or massive, respectively. If nothing is specified, the particle is considered as massive.

The Fortran name is defined by the keyword \texttt{Mass} and its default value by \texttt{MValue}. The value of keyword \texttt{MValue} is defined as a character string which is used in Fortran code.

The Fortran name of width and its default numerical value is given by keywords \texttt{Width} and \texttt{WValue}, respectively.
In the above definition, we do not assume any special name of particles; one can define particle name freely. However, some amplitude generation program is necessary to know some special particle appears or not. For example, CHANEL library offers calculation of amplitudes in general covariant gauge. In unitary gauge, \( \chi \)-scalars disappears from the calculation and one must drop Feynman graphs including them. In order to detect such kind of particles, particularly in the amplitude generating program, we add another keyword PCode with integer particle code. The values of PCode are not used in the Feynman graph generator.

### C.2 Definition of vertices

A definition of a particle begins with keyword `Vertex` and ends with keyword `Vend`.

The keyword `Vertex` defines the interacting particles in the list of their name. The direction of particle is defined as incoming to the vertex.

```plaintext
Vertex=\{\text{mu-e-bar, electron, W-plus}\};
```

defines a vertex which e\(^{-}\) and W\(^{+}\) come in and \( \nu_{e} \) goes out.

The order of coupling constants is given by keywords defined in the value of keyword `Order`.

```plaintext
ELWK=1;
```

A program generating amplitudes will require additional information. The name of coupling constant used in Fortran code of the vertex are given by `FName` keyword, whose data type and default value are given by the `FValue` keyword.

```plaintext
FName=ZW\(W\); FValue=\{\text{R, " GG*GCOS"}\};
```

where, "R" in the value of `FValue` represents that variable `ZW\(W\)` is an real number (or zero imaginary part). The definition of coupling constant " GG*GCOS" will be used in the FORTRAN source code generator to calculate the numerical value of the coupling constant.

One can define a counter term of renormalization as a vertex with higher order coupling constants.
D Output file

Here we describe the format of output file. We show an example of the output file for the input file shown in appendix B.

```plaintext
Model="all.mdl";

Process=1;
External=4;
OPI=No;Expand=Yes;

Graph=1;
Sfactor=1;
Vertex=4;

0={ 1[positron]};
1={ 2[electron]};
2={ 3[w-plus]};
3={ 4[w-minus]};

4[order={1,0}]={ 1[electron], 2[positron], 5[photon]};
5[order={1,0}]={ 5[photon], 6[w-minus], 7[w-plus]};
6[order={1,0}]={ 3[w-minus], 6[w-plus], 8[z]};
7[order={1,0}]={ 4[w-plus], 7[w-minus], 8[z]};

Vend;
Gend;

Graph=316;
Sfactor=1;
Vertex=3;

0={ 1[positron]};
1={ 2[electron]};
2={ 3[w-plus]};
3={ 4[w-minus]};
```

...
4[order={1,0}]= { 1[electron], 4[w-plus], 5[nu-e-bar]};
5[order={1,0}]= { 2[positron], 3[w-minus], 6[nu-e]};
6[loop=1;order={2,0}]= { 5[nu-e], 6[nu-e-bar]};
Vend;
Gend;
%/---------------------------------------
Pend=316;
/*------------------------------*/
Process=2;
External=4;
  0= initial electron;
  1= initial positron;
  2= final w-plus;
  3= final w-minus;
Eend;
elwk=2;Loop=0;
OPI=No;Expand=Yes;
%/---------------------------------------
Graph=1;
...

Graph=28;
Sfactor=1;
Vertex=2;
  0={ 1[positron]};
  1={ 2[electron]};
  2={ 3[w-plus]};
  3={ 4[w-minus]};
  4={ 5[photon]};
5[order={1,0}]= { 1[electron], 2[positron], 6[z]};
6[order={2,0}]= { 3[w-minus], 4[w-plus], 5[photon], 6[z]};
Vend;
Gend;
%/---------------------------------------
Pend=28;
/*------------------------------*/
End=3;

Lines beginning with “%” are comment lines.
The output file is an sequence of statements. Almost statements are written in the format:

“keyword=value” or “keyword”.

33
separated by colons “;”. The keyword part is an identifier (name composed of alphabet and digits). The value part is either an identifier or a list of identifiers separated by “,” and enclosed by braces “{" and “}”. An identifier may have options enclosed by brackets “[“ and “]”. Between two brackets, a statement or a list of statements is placed.

The first non-comment line specifies the file name of model definition which is copied from the input process file.

The output file includes several processes in accordance with the input file. In this example, the output file includes three processes. Each of them begins with the line:

\[
\text{Process}=<\text{process number}>;
\]

and ends with the line:

\[
\text{Pend}=<\text{the number of graph in the process}> ;
\]

The description of a process composed of

1. Description of external particles.
2. Options for graph generation.
3. Description of generated graphs.

The description of external particles begins with the statement:

\[
\text{External}=<\text{the number of external particles}> ;
\]

and then description of each external particle in the format

\[
<\text{external particle number}> = \text{initial} <\text{particle name}> ;
\]

or

\[
<\text{external particle number}> = \text{final} <\text{particle name}> ;
\]

The part \(<\text{external particle number}>\) is integer number beginning from 0. The last statement of the description of external particles is

\[
\text{Eend};
\]

The description of a generated graph begins with the statement

\[
\text{34}
\]
Graph=<graph number>;
and ends with the statement:
Gend;
After the Graph statement, it follows the global factor of the graph:
Sfactor=<inverse of global factor>;
The value of Sfactor is the inverse of the product of symmetric factor of the graph and sign produced by permutating external fermions and fermion loops.
The main part of the description of the graph begins with:
Vertex=<the number of generated vertices>;
and end with:
Vend;
Between these two statements, statements appear corresponding to nodes (external particles and vertices) including information how they are connected. The description of node is

<node number>=<internal line number>[particle name];
The <node number> is sequentially numbered beginning from 0. When multiple coupling constants are defined in the model, the following option is added to the <node number>:

order={order-1, order-2, ..., order-n}
which specifies the orders of coupling constants, corresponding to the Order statement in the model file. Moreover, when the vertex is not a tree one, the following option is also added

loop=<the number of loops>
The <internal line number> is numbered sequentially beginning from 1. Two nodes, which have common <internal line number>, are connected by the internal line. The <particle name> is the name of particle assigned to the internal line defined as particle and anti-particle for incoming and outgoing particle to the node, respectively.
The file end with the statement:
End=<the number of processes>;