`grcc` : a Feynman graph generator

T. Kaneko

2019.01.10

Contents

1 Changes from older version

Changes from ver. 2018.12.25

- Bugs found by `valgrind` are fixed.
- Global functions are changed to static and confined within `"grcc.cc"`.
- A switch '`defpart`' is added in struct `MInput` for the definition of models. When this switch is set to '`GRCC_DEFBYNAME`', particles and interactions are identified by character string name. When it is set to '`GRCC_DEFBYCODE`', particles and interactions are identified by integer code, and one can define a model without particle nor interaction names.

  With this change some data types are changed.
- Header file `"grcc.h"` is divided into `"grcc.h"` and `"grccparam.h"`.

  File `"grccparam.h"` consists of system parameters and definitions of some types, which can be used in C program.
- `Namespace` '`Grcc`' can be used with

  ```
  #define GRCC_NAMESPACE
  ```

  at the beginning of `"grccparam.h"`.
- All names of macro variables are changed to have prefix '`GRCC_`'.
- Some variables used for options are moved to the array '`values`' and functions `setStep` and `set1PI` are removed.
- An option is added to register a user defined error-exit function.
- An example program is prepared calling `grcc` from C program.

`testfromc.c`	: the main program written in C.
`grccfromc.cc`	: interface functions written in C++.
`grccfromc.h`	: for communication between above two files.

  No particle nor interaction names are used in this example.

2 Introduction

This document explains the usage of program `grcc` of Feynman graph generation in field theories.

2.1 Flow of the Feynman graph generation

Program `grcc` generates the set of Feynman graphs in accordance with the given orders of coupling constants and the set of external particles.

Before explaining the Feynman graph generation, we fix terms for graphs. A *graph* consists of *nodes* and *edges*. A node represents a vertex or an external particle. An edge is a line connecting two nodes. A node has a fixed number of legs, slots to be connected to edges. Let us call this number of legs *degree* of a node. An edge is a line connecting two legs. These legs may belong to a node or to two different nodes. A model defines a set of particles and a set of interactions. In a Feynman graph, a node is attributed to one of interactions or to an external particle, and an edge is attributed to one of particles defined in a given model. An edge has a direction along that momentum and other quantum numbers flow. At a leg of a node these quantum numbers are assumed coming into the node. We call such a Feynman graph an `agraph` here. When we look at only how nodes and edges are connected, we obtain more abstract and simpler graphs, forgetting about momenta, quantum numbers and directions of edges. We call such a graph a `mgraph`, that represents the structure of topological property of an `agraph`.

Program `grcc` generates first connected `mgraph`s without duplication, then assigns particle to each edge and an interaction to each node and then the graph becomes an `agraph`. Logically one can consider that `grcc` enumerates all possibilities of assignment, and then delete duplicated graphs.

2.2 Methods

The basic algorithms of graph generation and particle assignment are the same ones of program `grc`[1].

1. Graph Isomorphism

   Duplicated graphs are eliminated by orderly algorithm, developed by graph theorists, as described in Ref.[2]

2. Acceleration

   The main acceleration method is to limit the permutation group of nodes needed in the isomorphism test of graphs to its subgroup, same as program `grc` described in [1].

3. Structure of graphs

   The method of analysis of topologically invariant properties including connectivity, one-particle irreducibility, etc. is based on the algorithm of biconnectivity analysis described in Ref.[3].

4. Particle assignment

   The method of assigning particles and interactions is keeping a list of candidates for each edge and node, and updating these lists dynamically.

Throughout the graph generation procedure, back-tacking methods is applied to enumerate all possible configurations.

2.3 Differences from `grc`

The main differences between `grc` and `grcc` are:

1. Vacuum graphs can be generated in `grcc` but not in `grc`.
2. Grouping of external particles as topologically identical ones is possible in `grcc` but not in `grc`. For example, two graphs of gluon selfenergy graphs may be regarded topologically same when two external gluons are exchanged. It is possible in `grcc`, while is impossible in `grc`.
3. One obtains a set M_1 of `mgraph`s (topologies) by skipping particle assignment step. On the other hand one can also construct the set M_2 of topologies extracting from `agraphs` (particle-assigned graphs) by ignoring particles and interactions. In `grc` these two sets of topologies are different and the set M_2 contains isomorphic topologies. In the case of `grcc` these two sets are identical. It is realized by applying isomorphism tests to both `mgraph`s and `agraph`s. It requires the cost of a systematic acceleration method used in `grc` cannot be applied to `grcc`.
4. `grcc` is written in C++ while `grc` in C language.
5. Except for some special cases, `grcc` is slower than `grc` up to around factor 2.
6. `grcc` consumes more memory than `grc`. In the current parameter settings, `grcc` uses around 13 MBytes, while `grc` consumes less than 5 MBytes.

2.4 Files in this package

This package contains the following files

- Body of `grcc`

`grcc.h`	Header file defining classes.
`grccparam.h`	Header file defining parameters and some structs
`grcc.cc`	Code of functions

- Sample model definitions

`models.h`	Header file
`models.cc`	Definitions of models.

- Test programs

`testmodel.cc`	Simple program to test `"models.cc"`.
`testmgraph.cc`	Testing `mgraph` (topology) generation
`phicount.cc`	Table of expected numbers of `mgraph`s used in `"testmgraph.cc"`.
`testproc.cc`	Simple program of generating `agraph`s.
`testass0.cc`	Simple program of partial generation of `agraph`s.
`testass.cc`	Test program of full and partial generation of `agraph`s.
`testgrcc.cc`	Test program for multiple processes comparing with the results of `grc`.
`test1.cc`	Test program of non-standard model.

- Sample program called by C program

`testfromc.c`	C program calling `grcc`.
`grccfromc.cc`	Interface between `"testfrom.c"` and `grcc`.
`grccfromc.h`	Header file shared by `"testfrom.c"` and `"grccfrom.c"`.

- Results of performance test

`results/qcd1.grcc.result`	Performance in QCD with one quark.
`results/stand.grcc.result`	Performance in the standard model.

3 Usage

One can generate the set of all Feynman graphs or its subset in one of the following three ways.

1. Full generation

   Generate the set of all Feynman graphs (`agraph`s) corresponds to a given physical process. A process is defined by the set of initial and final particles, and the orders of coupling constants (a model may have more than one coupling constants like the standard model.) Let us call this way of generation *full generation*, and its input `process`.

2. Partial generation

   Generate a subset of the set of all Feynman graphs for a process. The input specifies the orders of coupling constants, and the set of initial and final particles and information about the set of vertices. Vertices are classified into groups by the total order of coupling constants and the number of legs. One has to specify how many vertices in each group should appear in the graphs. Let us call this way of generation *partial generation*. and its input `subprocess`. Full generation is constructed as a sequence of partial generations.

3. Topology generation

   Generate only topologies (`mgraph`) for a given set of nodes. Let us call this way of generation *topology generation*.

Information of a generated graph, for both `mgraph` and `agraph`, is represented by a set of data (let us call it `egraph`) as an object of class `EGraph`.

Example programs of Feynman graph generation are shown in `"testassign.cc"` for full and partial generation and `"testmgraph.cc"` for topology generation. See also `"testgrcc.cc"` for full generation of multiple processes.

At first we describe the procedure for partial generation. It proceeds as the following:

1. Setting up the model in which Feynman graphs are generated.
2. Setting up options.
3. Definition `subprocess`.
4. Creation of an `SProcess` class object and call `generate()` function of this class.
5. Finalization.

■ **Example**

Let us look at an example (`"testass0.cc"`):

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "grcc.h"
5 #include "models.h"
```

```
 6
 7 #ifdef GRCC_NAMESPACE
 8 using namespace Grcc;
 9 #endif
10
11 //=============================================================
12 // example of functions to be called when a new graph is generated.
13
14 static BigInt mgcount = 0;
15 static BigInt agcount = 0;
16
17 //-------------------------------------------------------------
18 void funcMG(EGraph *eg, void *pt)
19 {
20      mgcount++;
21      printf("+++ funcMG:%8ld           (%8ld, pt=%p)\n",
22             mgcount, eg->mId, pt);
23 }
24
25 //-------------------------------------------------------------
26 void funcAG(EGraph *eg, void *pt)
27 {
28      agcount++;
29      printf("+++ funcAG:%8ld, %8ld (%8ld, pt=%p)\n",
30             mgcount, agcount, eg->aId, pt);
31 }
32
33 //-------------------------------------------------------------
34 void erExit(const char *msg, void *pt)
35 {
36      printf("*** Error %s: exit program (%p) ***\n", msg, pt);
37      exit(1);
38 }
39
40 //-------------------------------------------------------------
41 Options *setOpt(Model *model)
42 {
43     Options *opt;
44
45     // options
46     opt = new Options();
47
48     // graph selection
49     // opt->values[GRCC_MGraph];                // only topology
50     // opt->values[GRCC_OPT_NoTadpole]  = True;  // without tadpoles
51     // opt->values[GRCC_OPT_NoTadpole]  = False; // with tadpoles
52     opt->values[GRCC_OPT_1PI]        = True;     // only 1PI graphs
53     // opt->values[GRCC_OPT_1PI]        = False; // connected graph
54
55     // output options
56     opt->setOutput(True, "out.grf");   // output to 'out.grf'
57     // opt->setOutput(False, "");      // no output file
58     opt->setOutMG(funcMG, NULL);       // for each mgraph
59     opt->setOutAG(funcAG, NULL);       // for each agraph
```

```
60      opt->setErExit(erExit, NULL);      // for error exit
61      opt->printLevel(2);                // print messages
62
63      // print starting message
64      opt->begin(model);
65
66      // print model information to a file
67      opt->outModel();
68
69      return opt;
70  }
71
72  //-------------------------------------------------------------
73  void qcdtest0(void)
74  {
75      Options  *opt;
76      Model    *model;
77      SProcess *sproc;
78
79      int j, defpart;
80      int gluon;
81      int quark, qbar, gghost;
82
83      int cdeg[GRCC_MAXNODES], ctyp[GRCC_MAXNODES];
84      int ptcl[GRCC_MAXNODES], cnum[GRCC_MAXNODES];
85      int cple[GRCC_MAXNODES];
86      int clist[GRCC_MAXNCPLG];
87      int ninitl, nfinal, n3, n4, nc;
88
89      // model
90      defpart = GRCC_DEFBYNAME;
91      model = modelQCD1(defpart);      // import a model
92      model->prModel();                // print the model
93
94      // options
95      opt = setOpt(model);
96
97      // get internal codes of particles
98      if (model->defpart == GRCC_DEFBYCODE) {
99          gluon  = model->findParticleCode( GRCC_GLUON);
100         quark  = model->findParticleCode( GRCC_QUARK);
101         qbar   = model->findParticleCode(-GRCC_QUARK);
102         gghost = model->findParticleCode( GRCC_GHOST);
103     } else {
104         gluon  = model->findParticleName("gluon");
105         quark  = model->findParticleName("q");
106         qbar   = model->findParticleName("q-bar");
107         gghost = model->findParticleName("c-gho");
108     }
109
110     printf("Particles\n");
111     printf("    gluon=%d, quark=%d, qbar=%d, gghost=%d\n",
112             gluon, quark, qbar, gghost);
113
```

```
114     // the number of initial and final particles and vertices
115     ninitl = 2;
116     nfinal = 2;
117     n3     = 4;
118     n4     = 1;
119
120     // order of coupling constant
121     for (j = 0; j < model->ncouple; j++) {
122         clist[j] = 0;
123     }
124     clist[0] = n3 + 2*n4;
125
126     // construct input arrays
127     nc = 0;
128     if (ninitl > 0) {
129         for (j = 0; j < ninitl; j++) {
130             cdeg[nc] = 1;
131             ctyp[nc] = GRCC_AT_Initial;
132             ptcl[nc] = gluon;
133             cple[nc] = 0;
134             cnum[nc] = 1;
135             nc++;
136         }
137     }
138     if (nfinal > 0) {
139         for (j = 0; j < nfinal; j++) {
140             cdeg[nc] = 1;
141             ctyp[nc] = GRCC_AT_Final;
142             ptcl[nc] = gluon;
143             cple[nc] = 0;
144             cnum[nc] = 1;
145             nc++;
146         }
147     }
148     if (n3 > 0) {
149         cdeg[nc] = 3;
150         ctyp[nc] = GRCC_AT_Vertex;
151         ptcl[nc] = 0;
152         cple[nc] = 1;
153         cnum[nc] = n3;
154         nc++;
155     }
156     if (n4 > 0) {
157         cdeg[nc] = 4;
158         ctyp[nc] = GRCC_AT_Vertex;
159         ptcl[nc] = 0;
160         cple[nc] = 2;
161         cnum[nc] = n4;
162         nc++;
163     }
164
165     // create SProcess
166     sproc = new SProcess(model, NULL, opt, 0, clist, nc,
167                          cdeg, ctyp, ptcl, cple, cnum);
```

```
168     sproc->prSProcess();    // print sprocess
169
170     // generate Feynman graphs
171     sproc->generate();
172
173     // end of the generation : print the result
174     opt->end();
175
176     delete sproc;
177     delete model;
178     delete opt;
179 }
180
181 //============================================================
182 int main(void)
183 {
184     qcdtest0();
185
186     return 0;
187 }
```

This program is compiled together with "grcc.cc" and "models.cc".

3.1 Include file

The header file for grcc is "grcc.h" and the header file of sample models is "models.h":

```
  4 #include "grcc.h"
  5 #include "models.h"
```

3.2 Namespace

When macro variable 'GRCC_NAMESPACE' is defined at the beginning of "grccparam.h":

```
#define GRCC_NAMESPACE
```

namespace 'Grcc' is used for all classes defined in grcc.

To absorb difference between cases whether this namespace is used or not, the following code is added:

```
  7 #ifdef GRCC_NAMESPACE
  8 using namespace Grcc;
  9 #endif
```

3.3 Setting up the model

The main part of the program is function 'qcdtest0'. We first look at this function.

A physical model used here is defined as an object of class Model. The following models are prepared in "models.h" and "models.cc":

`Model *modelPhi34(int);`	$\phi^3 + \phi^4$ model
`Model *modelQCD1(void);`	QCD with 1 quark
`Model *modelStandard(void);`	standard model

These functions return a pointer to the object of a model.

```
 76     Model    *model;
        ...
 90     defpart = GRCC_DEFBYNAME;
 91     model = modelQCD1(defpart);      // import a model
 92     model->prModel();                // print the model
```

The line '`defpart = GRCC_DEFBYNAME`' means that particles and interactions are identified by their name of character string. When '`defpart`' takes the following value:

```
        defpart = GRCC_DEFBYCODE;
```

then particles and interactions are identified by integer codes in stead of character string. See §3.5 for detail.

3.4 Setting up options

There are several options for graph generation. In this example, options are set in function '`setOpt`.

```
 95     opt = setOpt(model);
```

It will be explained later in §3.9.

3.5 Particle identification

Inside `grcc` particles are identified by internal codes. As identifiers of particles outside of `grcc` one can select either user defined integer codes or names in character string. This selection is specified by the value of '`defpart`' in

```
 91     model = modelQCD1(defpart);      // import a model
```

This value is saved in the model data and is accessible through `model->defpart`.

If the value of `defpart` value is '`GRCC_DEFBYCODE`' then particles are identified by integer codes, and if its value is '`GRCC_DEFBYNAME`' then particles are identified by names in character string. The following part of program gets internal particle codes and saved them in integer variables '`gluon`', '`quark`' etc.:

```
 98     if (model->defpart == GRCC_DEFBYCODE) {
 99         gluon  = model->findParticleCode( GRCC_GLUON);
100         quark  = model->findParticleCode( GRCC_QUARK);
101         qbar   = model->findParticleCode(-GRCC_QUARK);
102         gghost = model->findParticleCode( GRCC_GHOST);
103     } else {
104         gluon  = model->findParticleName("gluon");
105         quark  = model->findParticleName("q");
106         qbar   = model->findParticleName("q-bar");
107         gghost = model->findParticleName("c-gho");
108     }
```

Macro variables '`GRCC_GLUON`', '`GRCC_QUARK`', and '`GRCC_GHOST`' are defined in `"models.h"`.

3.6 Definition of the `subprocess`

A `subprocess` consists of

1. the orders of coupling constants,
2. the set of initial and final particles, and
3. information about the set of vertices.

■ Orders of coupling constants

Some models have two or more coupling constants. For example, the standard model has QCD and electro-weak coupling constants. The orders of coupling constants in the subprocess are specified by an array `clist` in this example. As QCD has one coupling constant, we use only the first component `clist[0]`:

```
124     clist[0] = n3 + 2*n4;
```

■ Grouping nodes

Nodes are classified into groups, whose members are considered topologically equivalent and then at least they have the same degree and the same total order of coupling constants. `grcc` generates permutations among nodes that keep these groups of nodes invariant. And then `grcc` picks up a representative from each set of graphs transformable by these permutations. Thus the number of graphs depends on how nodes are put into groups.

For example, if two identical initial particles are put into one group, and if two graphs are identical under the exchange of two initial particles, one of these two graphs is discarded. If these initial particles are put into different groups, such exchange of initial particles are no considered.

Vertices in a group should have the same number of legs (degree) and the total order of coupling constants. When there are multiple coupling constants in the model, division of total order to each coupling constant is made in all possible ways inside `grcc`.

A node corresponding to an external particle is either initial or final. Such a node has only one leg and particle is specified in the input. Particles are considered as in-coming to the graph for both initial and final (although final particles are considered out-going in full generation.) There are no difference between initial or final for partial generation. This information may be used in amplitude generation etc. after graphs are generated. If you don't want to distinguish between initial and final, you can use only "initial" for all external particles.

The following example deals the case of 2 initial and 2 final particles with 4 of 3-point vertices and 1 of 4-point vertices. The all external particles are considered topologically different each other. All vertices are assumed tree ones, that is, the order of coupling constant is 1 for 3-point and 2 for 4-point vertices.

```
114     // the number of initial and final particles and vertices
115     ninitl = 2;
116     nfinal = 2;
117     n3     = 4;
118     n4     = 1;
119
```

```
120     // order of coupling constant
121     for (j = 0; j < model->ncouple; j++) {
122         clist[j] = 0;
123     }
124     clist[0] = n3 + 2*n4;
125
126     // construct input arrays
127     nc = 0;
128     if (ninitl > 0) {
129         for (j = 0; j < ninitl; j++) {
130             cdeg[nc] = 1;
131             ctyp[nc] = GRCC_AT_Initial;
132             ptcl[nc] = gluon;
133             cple[nc] = 0;
134             cnum[nc] = 1;
135             nc++;
136         }
137     }
138     if (nfinal > 0) {
139         for (j = 0; j < nfinal; j++) {
140             cdeg[nc] = 1;
141             ctyp[nc] = GRCC_AT_Final;
142             ptcl[nc] = gluon;
143             cple[nc] = 0;
144             cnum[nc] = 1;
145             nc++;
146         }
147     }
148     if (n3 > 0) {
149         cdeg[nc] = 3;
150         ctyp[nc] = GRCC_AT_Vertex;
151         ptcl[nc] = 0;
152         cple[nc] = 1;
153         cnum[nc] = n3;
154         nc++;
155     }
156     if (n4 > 0) {
157         cdeg[nc] = 4;
158         ctyp[nc] = GRCC_AT_Vertex;
159         ptcl[nc] = 0;
160         cple[nc] = 2;
161         cnum[nc] = n4;
162         nc++;
163     }
```

The necessary information of each group are specified by the following items:

`cdeg[nc]`	the degree (the number of legs) of the node.
`ctyp[nc]`	type of the node.
`ptcl[nc]`	internal code of external particles.
`cple[nc]`	total order of coupling constants for vertices.
`cnum[nc]`	the number of nodes in this group

The value of "`ctyp[nc]`" is one of

`GRCC_AT_Vertex`	vertex
`GRCC_AT_Initial`	initial particle
`GRCC_AT_Final`	final particle

These group information are saved into arrays as in the example program.

If you want to regard all final particles as topologically identical, you will use the following code with "`cnum[nc] = nfinal;`":

```
if (nfinal > 0) {
    cdeg[nc] = 1;
    ctyp[nc] = GRCC_AT_Final;
    ptcl[nc] = gluon;
    cple[nc] = 0;
    cnum[nc] = nfinal;
    nc++;
}
```

3.7 Generation of Feynman graphs

With the information prepared above we create an object `sproc` of class `SProcess`. And then function `generate()` generates Feynman graphs.

```
165     // create SProcess
166     sproc = new SProcess(model, NULL, opt, 0, clist, nc,
167                          cdeg, ctyp, ptcl, cple, cnum);
168     sproc->prSProcess();    // print sprocess
169
170     // generate Feynman graphs
171     sproc->generate();
```

The arguments to the constructor `SProcess` are:

1	`model`	Pointer to the `Model` object.
2	`NULL`	This is used in full generation.
3	`opt`	Pointer to the `Options` object.
4	`0`	any identifying number of this subprocess.
5	`clist`	array of the order of coupling constants.
6	`nc`	the number of groups of nodes.
7	`cdeg`	array of degrees of groups.
8	`ctyp`	array of types of groups.
9	`ptcl`	array of internal particle code of external nodes.
10	`cple`	array of the total order of coupling constants of vertices.
11	`cnum`	array of the number of nodes in each group.

3.8 Finalization

After the generation, the following function is to be called:

```
174     opt->end();
```

3.9 Options

An object of class `Option` keeps options of graph generation and is also used to control the process of graph generation.

1. To generate only `mgraph`s (topologies) or `agraph`s (particle assigned graphs).

2. Conditions of graph selections; one-particle irreducible graphs, with or without tadpoles, etc.

3. To control the output of the program; how many messages being printed, writing to a file, etc. One can define output function by yourself which is called every time a new `mgraph` or `agraph` is generated.

4. For error-handling.

These options are set in function '`setOpt`' in this example.

```
43     Options *opt;
44
45     // options
46     opt = new Options();
47
48     // graph selection
49     // opt->values[GRCC_MGraph];                 // only topology
50     // opt->values[GRCC_OPT_NoTadpole]  = True;  // without tadpoles
51     // opt->values[GRCC_OPT_NoTadpole]  = False; // with tadpoles
52     opt->values[GRCC_OPT_1PI]        = True;     // only 1PI graphs
53     // opt->values[GRCC_OPT_1PI]        = False; // connected graph
54
55     // output options
56     opt->setOutput(True, "out.grf");   // output to 'out.grf'
57     // opt->setOutput(False, "");      // no output file
58     opt->setOutMG(funcMG, NULL);       // for each mgraph
59     opt->setOutAG(funcAG, NULL);       // for each agraph
60     opt->setErExit(erExit, NULL);      // for error exit
61     opt->printLevel(2);                // print messages
```

■ **Creation of `Options` object**

At first, an object of `Options` class is created by

```
46     opt = new Options();
```

As this object has a role to control the program, one cannot skip to create this object in a program.

■ **`mgraph` or `agraph` generation**

Generate only `mgraph`s by

```
opt->values[GRCC_MGraph];
```

For `agraph` generation (this is default):

```
opt->values[GRCC_AGraph];
```

■ **Graph selection**

Program `grc` generates connected graphs and then some categories of graphs are eliminated according to specified options:

```
50     // opt->values[GRCC_OPT_NoTadpole]  = True;  // without tadpoles
51     // opt->values[GRCC_OPT_NoTadpole]  = False; // with tadpoles
52     opt->values[GRCC_OPT_1PI]        = True;     // only 1PI graphs
53     // opt->values[GRCC_OPT_1PI]        = False; // connected graph
```

The available options are:

`OPT_1PI`	generate only 1PI graphs
`OPT_NoTadpole`	exclude graphs with tadpoles
`OPT_No1PtBlock`	exclude graphs with tadpole blocks
`OPT_No2PtL1PI`	exclude graphs with 2-point subgraphs

A graph with "tadpole block" means that the graph contains a subgraph which becomes disconnected bubble without external particles when the graph is cut at one vertex.

When all of them are set to `False`, `grcc` generates all connected graphs.

When two or more options of graph selection are specified to `True`, it is equivalent logically to:

```
(a connected graph is generated)
if (opt->values[OPT_NoTadpoles])
        if (graph has one or more tadpole)
                discard graph
if (opt->values[OPT_1PI])
        if ( graph is NOT one-particle irreducible )
                discard graph
    ...
```

(It is noted that a one-particle irreducible graph has no tadpoles.)

■ Output to a file

`grcc` includes a utility to produce a file which is compatible to use `grcdraw` (graph drawer) and `grcplot` (graph plotter) included in `grc.v2.2` [1].

The line

```
56     opt->setOutput(True, "out.grf");   // output to 'out.grf'
57     // opt->setOutput(False, "");      // no output file
```

specifies to produce an input file for these utilities. These utilities requires the model file consistent with this output. The corresponding model file (`"QCD1.mdl"` in this case) is generated by calling:

```
75     opt->outModel();       // write to file "QCD1.mdl"
```

When the number of graphs is huge, `"out.grf"` becomes also huge.

■ Option for printing messages

The following line determines how many messages are printed.

```
61     opt->printLevel(2);                // print messages
```

When the argument is 0 instead 2, no messages will be printed.

■ Creating your own output

You can specify your own functions which are called when new graph is generated. These functions will be used for the purpose of amplitude generation,

saving graph information in files, etc. Here we show a simple example. More explanation will be given in section 4. Let us assume the following function is prepared:

```
18 void funcMG(EGraph *eg, void *pt)
19 {
20      mgcount++;
21      printf("+++ funcMG:%8ld           (%8ld, pt=%p)\n",
22             mgcount, eg->mId, pt);
23 }
```

This function just counts the number of graphs. When the following option is specified:

```
58     opt->setOutMG(funcMG, NULL);       // for each mgraph
```

Function `funcMG` will be called every time a new `mgraph` is generated. This function is called by `grcc` in the form:

```
       funcMG(egraph, NULL);
```

Similarly,

```
61     opt->setOutAG(funcAG, NULL);       // for each agraph
```

specifies the function which is called when a new `agraph` is generated.

■ **User defined error-exit function**

When an error is found and `grcc` thinks unable to continue, the `grcc` stops the program. It is possible to pass the control of the program to user defined function before program stops. The function is defined as (of course the name of the function can be changed):

```
34 void erExit(const char *msg)
```

and register to '`grcc`' by:

```
60     opt->setErExit(erExit, NULL);      // for error exit
```

The argument '`msg`' is an error message. See also §4.2 for details.

3.10 Full generation

Full generation of Feynman graph corresponds to the conditions of usual physical processes, which means:

- External particles are considered topologically different each other and each of them consists a group of node by itself.
- The initial particle is assumed in-coming to the graph and the final one is out-going from the graph.

  Attention! The convention of final particles is different from the case of partial generation.

■ **Example**

For full generation one has to specify the orders of coupling constants, and the sets of initial and final particles. An example program is:

```
 1 void qcdtest1(void)
 2 {
 3     // sample of partial generation
 4 
 5     Counter  counter;
 6     Options *opt;
 7     Model   *model;
 8     Process *proc;
 9     int      pid = 1;
10 
11     int initlPart[GRCC_MAXNODES], finalPart[GRCC_MAXNODES];
12     int cpl[GRCC_MAXNCPLG];
13     int ninitl, nfinal, loop, defpart;
14     int uq, ubar, gluon;
15 
16     // model
17     defpart = GRCC_DEFBYCODE;
18     // defpart = GRCC_DEFBYNAME;
19     model = modelQCD1(defpart);
20     model->prModel();
21 
22     // options
23     opt = setOpt(model, &counter);
24 
25     // get particle code
26     if (model->defpart == GRCC_DEFBYCODE) {
27         uq     = model->findParticleCode( GRCC_QUARK);
28         ubar   = model->findParticleCode(-GRCC_QUARK);
29         gluon  = model->findParticleCode( GRCC_GLUON);
30     } else {
31         uq     = model->findParticleName("q");
32         ubar   = model->findParticleName("q-bar");
33         gluon  = model->findParticleName("gluon");
34     }
35 
36     printf("q=%d, q-bar=%d, gluon=%d\n", uq, ubar, gluon);
37 
38     // initial and final particles
39     ninitl = 2;
40     nfinal = 1;
41 
42     initlPart[0] = gluon;
43     initlPart[1] = gluon;
44     finalPart[0] = gluon;
45 
46     // order of coupling constants
47     loop = 2;
48 
49     cpl[0] = 2*loop + ninitl + nfinal - 2;
50 
51     if (loop == 0) {
52         opt->values[GRCC_OPT_1PI] = False;     // for tree
53     }
54 
```

```
55     // create process object and generate Feynman graphs
56     proc = new Process(pid, model, opt,
57                        ninitl, initlPart, nfinal, finalPart, cpl);
58
59     // print the summary of the result
60     opt->end();
61
62     printf("mgcount = %ld, agcount = %ld\n",
63            counter.mgcount, counter.agcount);
64     fclose(counter.fp);
65
66     // delete created objects
67     delete proc;
68     delete opt;
69     delete model;
70 }
```

In the constructor of `Process` object, several `subprocess` objects are created and the function `sproc->generate()` is called for each of them. See also `qcdtest1` function in `"testass.cc"`.

3.11 Topology generation

For `mgraph` (topology) generation, create first an `MGraph` object by:

```
mgraph = new MGraph(pid, cc, cldeg, clnum, cltyp, opt);
```

Before calling this function, one has to define groups of nodes; nodes are put into groups of topologically equivalent ones.

■ **Arguments of 'new MGraph'**

Arguments are:

1. `pid` : Any integer used identifying the `mgraph` object.
2. `cc` : The number of groups of nodes.
3. `cldeg[j]` : the degree of nodes in the group `j` $(0 \leq \texttt{j} < \texttt{cc})$.
4. `clnum[j]` : the number of nodes in the group `j`.
5. `cltyp[j]` : the type of nodes in the group `j`. Its value is one of

-1 :	external particle
0 :	tree vertex
1 :	1 loop counter term or effective vertex
n > 0 :	n loop counter term or effective vertex

6. `opt` : `Options` object; same as in §3.9

■ **Graph generation**

After creating an object `mgraph`, function `generate()` of `MGraph` class generates graphs:

```
ncon = mgraph->generate();
```

This function returns the number of generated graphs.

4 Output function

4.1 Registration of user functions

Output is made by writing out the information kept in an `egraph` object as will be shown in §4.3. You can use your own output functions by registering them to `grcc` using `setOutMG` function of `Options` class.

For example, let us define data type `Counter` and function `funcMG` as the following:

```
typedef struct {
    FILE *fp;
    long  mgcount;
    long  agcount;
} Counter;

void funcMG(EGraph *eg, void *pt)
{
     Counter *pc = (Counter *)pt;

     if (pt == NULL) {
         printf("+++ funcMG: mId=%ld\n", eg->mId);
     } else {
         pc->mgcount++;
         fprintf(pc->fp, "+++ funcMG:%ld\n", pc->mgcount);
     }
}
```

Then we call `setOutMG` in the following way:

```
Options *opt;
Counter  counter;
    ...
opt = new Options();
    ...
if ((counter.fp = fopen("out.log", "w")) == NULL) {
    fprintf(stderr, "*** cannot open for output.\n");
    exit(1);
}
counter.mgcount = 0;
counter.agcount = 0;
opt->setOutMG(funcMG, (void *) &counter);
    ...
proc = new Process(...)
    ...
opt->end();
    ...
fclose(counter.fp);
printf("mgcount = %ld, agcount = %ld\n",
       counter.mgcount, counter.agcount);
```

`grcc` calls function `funcMG` for each `mgraph` in the form:

```
funcMG(egraph, pt);
```

where '`*pt = counter`'. With this program the numbers of graphs will be written on the file `"out.log"`. Similarly one can use `setOutputAG`. See also `"testass.cc"`.

4.2 Error handling

Error handling in `grcc` is very primitive.

1. Print message by '`sprintf(GRCC_Stderr, ...)`'.

   Macro variable '`GRCC_Stderr`' is '`stdout`' or '`stderr`' as defined in `"grccparam.h"`.

2. Call function '`erEnd(message)`'.

3. '`erEnd(message)`' prints message to '`GRCC_Stderr`' and call standard function '`abort()`' or '`exit(1)`' according to the definition of '`GRCC_ABORT()`'.

This control of the program is changed by specifying error-exit function as an option. First define a function with the following interface:

```
void erExit(const char *msg, void *pt);
```

Argument '`msg`' is an error-message and a pointer '`pt`' as same as function '`funcMG`' in §4.1. This function is registered to '`grcc`' by:

```
setErExit(erExit, (void *) pt);
```

■ **Example**

Let us consider the following function `erExit` in addition to the example in §4.1.

```
void erExit(const char *msg, void *pt)
{
     Counter *pc = (Counter *)pt;

     if (pt != NULL) {
         fclose(pc->fp);
     }
     printf("*** Error %s: exit program ***\n", msg);
     exit(1);
}
```

And register this function by:

```
opt->setErExit(erExit, (void *) counter);
```

Then the file pointer '`counter.pt`' is closed normally even when the program stops because of an error.

4.3 Accessing components of a graph

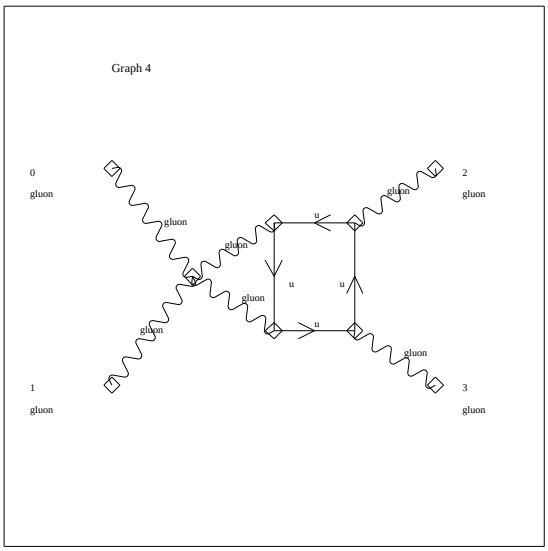

Figure 1: Plotted Feynman graph

In order to see how to access the information of a generated graph, let us look at an output function writing to file `"out.grf"`, as an example. File `"out.grf"` contains enough information to reconstruct the Feynman graph, and is used as input to `grcdraw` or `grcplot` in the package `grc.v2.2`[1], The plotted graph is shown in Fig.1. The corresponding part of the output file is shown below:

```
%-----------------------------
Graph=4;
% AGraph=4;
Gtype=1;
Sfactor=-1;
Vertex=5;
   0={   1[gluon]};
   1={   2[gluon]};
   2={   3[gluon]};
   3={   4[gluon]};
   4={   7[gluon],    5[u-bar],    6[u]};
   5={   9[gluon],    8[u-bar],    5[u]};
   6={   3[gluon],    6[u-bar],   10[u]};
   7={   4[gluon],   10[u-bar],    8[u]};
   8={   1[gluon],    2[gluon],    7[gluon],    9[gluon]};
%  FLines=1; FSign=-1; sId=4;
%    0[Loop]=[5, 8, 10, 6];
Vend;
```

```
Gend;
```

■ Output format

- `Graph=4;`

  means this graph is the 4th `agraph` as the sequential number in a process (or in a subprocess for a partial generation).

- `Gtype=1;`

  means this `agraph` is brought from the first `mgraph`.

- `% AGraph=4;`

  This line is a comment line as an input to `grcdraw`. It shows the sequential number of `agraph` within the set of `agraph`s generated from the current `mgraph`.

- `Sfactor=-1;`

  This line supplies information about the global factor of the graph. It is the denominator of the symmetry factor multiplied by the sign factor coming from fermi statistics. Of course this sign factor is determined up to global sign common to all graphs in a physical process.

  Attention! This sign factor is not calculated correctly for interactions with more than 2 fermions, like four-fermi interactions or some kind of effective interactions.

- `Vertex=5;`

  means that there are 5 vertices. There are 9 nodes in total with 4 external particles.

- `4={ 7[gluon], 5[u-bar], 6[u]};`

  means that node 4 is a vertex which `gluon`, `u-bar` and `u` come in, and each leg is connected to edges 7, 5 and 6, respectively. The opposite side of these edges are node 8 (3rd leg), 5 (3rd leg) and 6 (2nd leg), respectively; this relation is found from the lines of corresponding nodes.

- `% FLines=1; FSign=-1; sId=4;`

  shows that there is one fermion line in the graph and sign is -1, related to fermi statistics. These fermion lines are not defined in `mgraph`s, of course.

- `% 0[Loop]=[5, 8, 10, 6];`

  shows the sequence of edges on a fermion line. In this example there is only one looped fermion line, numbered starting from 0. It consists of edges 5, 8, 10, and 6, in this order. When there are external fermions in the process, lines like

  `% 0[Open]=[0, 3, 5, 1];`

  will appear.

■ Code producing `"out.grf"`

The code producing this data is function `Output::outEGraph` found in `"grcc.cc"`:

```
 1 void Output::outEGraph(EGraph *egraph)
 2 {
 3     int nd, lg, ptcl, ed, intr, j, k, loop;
 4     Model *mdl = egraph->model;
 5     Bool popt;
 6     EFLine *fl;
 7
 8     if (outgrfp == NULL) {
 9         return;
10     }
11     fprintf(outgrfp, "%%-----------------------------\n");
12     if (egraph->assigned) {
13         fprintf(outgrfp, "Graph=%ld;\n", egraph->sId);
14         fprintf(outgrfp, "%% AGraph=%ld;\n", egraph->aId);
15     } else {
16         fprintf(outgrfp, "Graph=%ld;\n", egraph->mId);
17     }
18     fprintf(outgrfp, "Gtype=%ld;\n", egraph->mId);
19     if (egraph->assigned) {
20         fprintf(outgrfp, "Sfactor=%ld;\n",
21                 egraph->nsym * egraph->esym * egraph->fsign);
22     } else {
23         fprintf(outgrfp, "Sfactor=%ld;\n",
24                 egraph->nsym * egraph->esym);
25     }
26     fprintf(outgrfp, "Vertex=%d;\n",
27             egraph->nNodes - egraph->nExtern);
28
29     for (nd = 0; nd < egraph->nNodes; nd++) {
30         fprintf(outgrfp, "%4d", nd);
31         if (mdl != NULL && egraph->assigned &&
32                 !egraph->isExternal(nd)) {
33             intr = egraph->nodes[nd]->intrct;
34             loop = mdl->interacts[intr]->loop;
35             popt = False;
36
37             // not tree vertex
38             if (loop > 0) {
39                 fprintf(outgrfp, "[loop=%d", loop);
40                 popt = True;
41             }
42
43             // multiple coupling constants
44             if (mdl->ncouple > 1) {
45                 if (popt) {
46                     fprintf(outgrfp, ",order={");
47                 } else {
48                     fprintf(outgrfp, "[order={");
49                     popt = True;
50                 }
51                 for (j = 0; j < mdl->interacts[intr]->nclist;
52                      j++) {
53                     if (j != 0) {
54                         fprintf(outgrfp, ",");
```

```
55                      }
56                      fprintf(outgrfp, "%d",
57                              mdl->interacts[intr]->clist[j]);
58                  }
59                  fprintf(outgrfp, "}");
60              }
61              if (popt) {
62                  fprintf(outgrfp, "]");
63              }
64          }
65          fprintf(outgrfp, "={");
66
67          // list of legs
68          for (lg = 0; lg < egraph->nodes[nd]->deg; lg++) {
69              if (lg != 0) {
70                  fprintf(outgrfp, ", ");
71              }
72              ed  = Abs(egraph->nodes[nd]->edges[lg])-1;
73              fprintf(outgrfp, "%4d", ed+1);
74              ptcl = egraph->edges[ed]->ptcl;
75              if (mdl != NULL && ptcl != 0 && egraph->assigned) {
76                  if (egraph->nodes[nd]->edges[lg] < 0) {
77                      ptcl = mdl->normalParticle(-ptcl);
78                  } else {
79                      ptcl = mdl->normalParticle(ptcl);
80                  }
81                  fprintf(outgrfp, "[%s]",
82                          mdl->particleName(ptcl));
83              } else {
84                  fprintf(outgrfp, "[undef]");
85              }
86          }
87          fprintf(outgrfp, "};\n");
88      }
89      // print Fermion line as comment line
90      fprintf(outgrfp, "%%  FLines=%d; FSign=%d; sId=%ld;\n",
91              egraph->nflines, egraph->fsign, egraph->sId);
92      for (j = 0; j < egraph->nflines; j++) {
93          fl = egraph->flines[j];
94          fprintf(outgrfp, "%% %4d", j);
95          if (fl->ftype == FL_Open) {
96              fprintf(outgrfp, "[Open]=[");
97          } else if (fl->ftype == FL_Closed) {
98              fprintf(outgrfp, "[Loop]=[");
99          } else {
100             fprintf(outgrfp, " ?%d", fl->ftype);
101         }
102         for (k = 0; k < fl->nlist; k++) {
103             if (k != 0) {
104                 fprintf(outgrfp, ", ");
105             }
106             fprintf(outgrfp, "%d", Abs(fl->elist[k]));
107         }
108         fprintf(outgrfp, "];\n");
```

```
109     }
110     fprintf(outgrfp, "Vend;\n");
111     fprintf(outgrfp, "Gend;\n");
112 }
```

■ Information of a graph

- `egraph->assigned` :

  When this `egraph` corresponds to a `mgraph` (`agraph`) then its value is `False` (`True`).

- `egraph->sId`, `egraph->mId`, `egraph->aId` :

  Identifier (sequential number) of `mgraph` is kept in `egraph->mId`. Variable `egraph->aId` represents the sequential number of the `agraph` generated from the `mgraph`. Variable `egraph->sId` is the sequential number counted from the beginning of the process (or subprocess for partial generation).

- `egraph->nsym`, `egraph->esym`, `egraph->fsign`:

  The denominator of symmetry factor of the graph is calculated by

  $$(\texttt{egraph->nsym}) \times (\texttt{egraph->esym}),$$

  where `egraph->nsym` and `egraph->esym` come from permutation of nodes and edges, respectively. Variable `egraph->fsign` represents the sign factor caused by fermion statistics (it is determined by the number of fermion loops and by the how fermion lines connect external fermions. This sign factor is determined up to global sign common to all generated graphs.)

- `egraph->nExtern`, `egraph->nNodes`:

  The number of external particles is `egraph->nExtern`, the number of nodes is `egraph->nNodes`. The number of vertices equals to

  $$(\texttt{egraph->nNodes}) - (\texttt{egraph->nExtern}).$$

- `egraph->model->ncouple` :

  The number of coupling constants defined in the model.

■ Information of a node

- `egraph->nodes[nd]->intrct` :

  The internal code of the interaction assigned to the `nd`th node in the graph.

- `egraph->model->interacts[intr]->clist[j]` :

  The list of the orders of coupling constants of the interaction defined in the model.

- `egraph->nodes[nd]->deg` :

  The degree of the node `nd`.

- `egraph->nodes[nd]->edges[lg]` :

  The value x of this variable indicates the edge connected to leg `lg` of node `nd`. An edge has a direction, represented by the sign of x, and the particle p flows along this direction on the edge. When $x < 0$ ($x > 0$), the particle p goes out from (comes into) the node. In other words, when $x < 0$ the anti-particle of the particle p comes into the vertex. The index of the table of edges starts from 0 and its value is obtained by $\mathrm{abs}(x) - 1$ (x is defined so as to avoid 0). Thus the object representing an edge is obtained by

  `egraph->edges[Abs(egraph->nodes[nd]->edges[lg])-1]`.

■ Information of an edge

- `egraph->edges[ed]->ptcl` :

  Internal code of particle flowing on the edge `ed`.

■ Information of the model

- `model->normalParticle(ptcl)` :

  Normalizing the sign of the internal particle code. When the particle is a real field then this function returns abs(`ptcl`), otherwise `ptcl`.

- `egraph->model->antiParticle(ptcl)`

  returns the $-$`ptcl` when the particle is a complex field. Otherwise it returns abs(`ptcl`).

- `egraph->model->particleName(ptcl)`.

  This function returns the character string corresponding to the name (character string) of particle or anti-particle. It returns the name of the anti-particle when `ptcl` < 0 and the particle is a complex field, and the name of the particle otherwise.

  If these names are not defined in the model, `grcc` will create some character string.

- `egraph->model->particleCode(ptcl)` :

  This function returns the integer code defined in the model corresponding to the internal code `ptcl` of the particle or anti-particle.

- `egraph->model->interacts[intr]->name` :

  The character string corresponding to the name of interaction `intr` defined in the model.

  If these names are not defined in the model, `grcc` will create some character string.

- `egraph->model->interacts[intr]->icode` :

  The integer code of `intr` defined in the model.

- `egraph->model->particles[ptcl]->ptype` :

  The type of the particle `ptcl`, which is one of

`PT_Undef`	Undefined
`PT_Scalar`	Scalar particle
`PT_Dirac`	Dirac particle
`PT_Majorana`	Majorana particle
`PT_Vector`	Vector boson
`PT_Ghost`	Ghost

- `egraph->model->particles[ptcl]->pcode` :

  The `pcode` of the particle code for `ptcl` (`ptcl` > 0).

■ Information of fermion lines

Let us call a path consisting of edges *fermion line*, when a fermion flows on the path. There are two kinds of fermion lines. One is fermion loop and the other connects two external fermions placed at two terminal points of the path. Let us call the former `closed` and the latter *open*.

These fermion lines are analysed by `grcc` and their information is acessible through follwoing variables:

- `egraph->nflines` :

  The number of fermion lines; sum of the numbers of loops and open fermion lines

- `egraph->flines[j]->ftype`:

  The type of `j`th fermion line, which value is `FL_Closed` or `FL_Open`.

- `egraph->flines[j]->nlist`:

  The number of edges on the `j`th fermion line.

- `egraph->flines[j]->elist[k]`;

  The `k`th edge on the `j`th fermion line. The sign of the value of this variable represents the direction of the edge.

5 Definition of a model

We show an example of definition of QCD with 1 quark (see "models.cc").

```
 1 Model *modelQCD1(int defprt)
 2 {
 3     Model *mdl;
 4     int j;
 5
 6     static MInput minp = {"QCD1", GRCC_DEFBYCODE, 1, {"QCD"}};
 7     static PInput plist[] = {
 8         {"gluon", "gluon",   GRCC_GLUON,  GRCC_GLUON,
                "vector", GRCC_PT_Vector},
 9         {"q",     "q-bar",   GRCC_QUARK, -GRCC_QUARK,
                "dirac",  GRCC_PT_Dirac},
10         {"c-g",   "c-g-bar", GRCC_GHOST, -GRCC_GHOST,
                "ghost", GRCC_PT_Ghost},
11     };
12     static IInput ilist[] = {
13         {"ggg",  GRCC_VGGG, 3,
14             {"gluon",      "gluon",    "gluon"},
15             { GRCC_GLUON,  GRCC_GLUON, GRCC_GLUON},             {1}},
16         {"qqg",  GRCC_VQQG, 3,
17             {"gluon",      "q-bar",    "q"},
18             { GRCC_GLUON, -GRCC_QUARK, GRCC_QUARK},             {1}},
19         {"gcgg", GRCC_VXXG, 3,
20             {"c-g-bar",    "c-g",      "gluon"},
21             {-GRCC_GHOST,  GRCC_GHOST, GRCC_GLUON},             {1}},
22         {"gggg", GRCC_VGGGG,4,
23             {"gluon",      "gluon",    "gluon",    "gluon"},
24             { GRCC_GLUON,  GRCC_GLUON, GRCC_GLUON, GRCC_GLUON}, {2}},
25     };
26     static int nplist = sizeof(plist)/sizeof(PInput);
27     static int nilist = sizeof(ilist)/sizeof(IInput);
28
29     minp.defpart = defprt;
30     mdl = new Model(&minp);
31     for (j = 0; j < nplist; j++) {
32         mdl->addParticle(plist+j);
33     }
34     mdl->addParticleEnd();
35
36     for (j = 0; j < nilist; j++) {
37         mdl->addInteraction(ilist+j);
38     }
39
40     mdl->addInteractionEnd();
41
42     // mdl->prModel();
43
44     return mdl;
45 }
```

A model is defined by the following 3 components:

1. Name of the model, how to identify particles and interactions, and definition of coupling constants.

2. Particles.

3. Interactions.

They are defined through the following `struct` data defined in "`grccparam.h`":

```
typedef struct {
    const char *name;               /* name of the model */
          int   defpart;            /* particle is defined by name or code */
                                    /* GRCC_DEFBYNAME or GRCC_DEFBYCODE */
          int   ncouple;            /* No. of coupling constants */
    const char *cnamlist[GRCC_MAXNCPLG]; /* Names of coupling constants */
} MInput;

typedef struct {
    const char *name;               /* name of particle */
    const char *aname;              /* name of anti-particle */
          int   pcode;              /* code of particle */
          int   acode;              /* code of anti-particle */
    const char *ptypen;             /* type : 'GRCC_PTS_Scalar' etc */
          long  ptypec;             /* type : 'GRCC_PT_Scalar' etc */
} PInput;

typedef struct {
    const char *name;                 /* name of interaction */
          int   icode;                /* code of interaction */
          int   nplistn;              /* the number of legs */
    const char *plistn[GRCC_MAXLEGS];    /* list of particle names */
          int   plistc[GRCC_MAXLEGS];    /* list of particle codes */
          int   cvallist[GRCC_MAXNCPLG]; /* coupling constants */
} IInput;
```

1. `MInput` : name of the model and coupling constants.
   (a) `name` : name of the model.
   (b) `defpart` : a particle or a vertex is identified by
      - integer code (=`GRCC_GRCC_DEFBYCODE`) or by
      - name (character string) (=`GRCC_GRCC_DEFBYNAME`).
   (c) `ncouple` : the number of coupling constants.
   (d) `cnamlist` : list of names of the coupling constants.

2. `PInput` : definition of a particle
   (a) `name` : name of the particle.
   (b) `aname` : name of the anti-particle.
   (c) `pcode` : code of particle. It should not be 0.
   (d) `acode` : code of anti-particle. It should not be 0.
   (e) `ptypen` : type of a particle represented in character string. Its value is one of:

```
GRCC_PTS_Scalar     =   "scalar"
GRCC_PTS_Dirac      =   "dirac"
GRCC_PTS_Majorana   =   "majorana"
GRCC_PTS_Vector     =   "vector"
GRCC_PTS_Ghost      =   "ghost"
```

(f) `ptypec` : type of a particle represented in an integer. Its value is one of:

`GRCC_PT_Scalar`	scalar
`GRCC_PT_Dirac`	dirac
`GRCC_PT_Majorana`	majorana
`GRCC_PT_Vector`	vector
`GRCC_PT_Ghost`	ghost

- When `defpart = GRCC_DEFBYCODE` in the model definition then
  - Particles and anti-particles are identified by `pcode` and `acode`.
  - If `pcode` = `acode` then the particle is considered a real field.
  - Type of the particle is defined by `ptypec`, and `ptypen` is ignored.
  - Items `name` and `aname` can be `NULL` or `""`, otherwise they should have meaningful values as these values may be used.
- When `defpart = GRCC_DEFBYNAME` in the model definition then
  - particles are identified by `name` and `aname`.
  - If `name` = `aname`, then the particle is considered a real field.
  - Type of the particle is defined by `ptypen`, and `ptypec` is ignored.
  - Items `pcode` and `acode` can be `0`, otherwise they should have meaningful values as these values may be used.

3. `IInput` : definition of an interaction

(a) `name` : name of the interaction.

(b) `icode` : integer code of the interaction. It should be a positive integer.

(c) `nplistn` : the number of interacting particles.

(d) `plistn[MAXLEGS]` : list of names of interacting particles

(e) `plistc[MAXLEGS]` : list of codes of interacting particles

(f) `cvallist[MAXNCPLG]` : list of coupling constants.

- When `defpart = GRCC_DEFBYCODE` in the model definition then
  - Interactions are identified by `icode`. The value of `plistc[j]` should be one of `pcode`s or `acode`s defined for particles.
  - `plistn[j]` is ignored.
- When `defpart = GRCC_DEFBYNAME` in the model definition then
  - Interactions are identified by `name`. The value of `plistn[j]` should be one of `name`s or `aname`s defined for particles.
  - `plistn[c]` is ignored.

We assume the usual relation between the total number of coupling constants C and the number of interacting particles $N =$ `pnlistn` in each interaction:

$$L = \frac{C - N + 2}{2} \qquad \text{is non-negative integer.}$$

L is considered as the number of loops included inside the interaction, which may be a counter term or an effective interaction.

Attention!

If there is an interaction with more than 2 fermions, such as four-fermi interactions or some kind of effective interactions, sign factor corresponding to fermi statistics is NOT correctly calculated in `egraph->fsign`. In this case, you have to calculate it in your output function or in amplitude generation.

6 Parameters

■ **Namespace**
A macro variable

```
#define GRCC_NAMESPACE
```

is defined in header file "grccparam.h" and all classes in grcc are put into namespace 'Grcc'.

■ **Parameters**
Program grcc has following parameters defined in "grccparam.h", that determine the maximum sizes of arrays. This header file is included in "grcc.h" and can be used in C and C++ programs.

GRCC_MAXNCPLG	The maximum number of coupling constants in the model.
GRCC_MAXLEGS	The maximum number of legs of interactions.
GRCC_MAXMPARTICLES	The maximum number of particles defined in the model.
GRCC_MAXMINTERACT	The maximum number of interactions defined in the model.
GRCC_MAXSUBPROCS	The maximum number of sub-processes in a process.
GRCC_MAXNODES	The maximum number of nodes in a graph.
GRCC_MAXEDGES	The maximum number of edges in a graph.
GRCC_MAXNSTACK	Node-stack size used in generation of agraphs.
GRCC_MAXESTACK	Edge-stack size used in generation of agraphs.
GRCC_MAXGROUP	The number of elements of the symmetry group of a graph.
GRCC_MAXPSLIST	The size of candidate list.

If some error messages appears with one of these names, you need to enlarge the value of the corresponding variable in "grccparam.h".

■ **Error handling**
The following macros are used for error handling:

GRCC_Stderr	This will be stdout or stderr to which error messages are printed.
GRCC_ABORT()	This will be exit(1) or abort() which will be called when program stops with errors.

7 Calling grcc from a C program

Here we show an example of calling grcc from a C program. This example consists of three files:

testfromc.c	main program written in C.
grccfromc.cc	interface functions to grcc written in C++.
grccfromc.h	header file used between above two programs.

7.1 C program

The main program of this example is written in C program "testfromc.c".

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #include "grccparam.h"
 5 #include "grccfromc.h"
 6
 7 #define GLUON 1
 8 #define GHOST 2
 9 #define QUARK 3
10
11 /*============================================================
12  */
13 int main(void)
14 {
15     int loop, j;
16
17     /* definition of a model */
18     /*   {name, how_prtcls_are_defined, no_c_consts, {c_consts}}
19      */
20     static MInput minp = {"QCD1", GRCC_DEFBYCODE, 1, {"g"}};
21
22     /* definition of particles:
23      *   {pname, aname, pcode, acode, typec, typen}
24      */
25     static PInput pinp[] = {
26         {NULL, NULL, GLUON,  GLUON, NULL, GRCC_PT_Vector},
27         {NULL, NULL, QUARK, -QUARK, NULL, GRCC_PT_Dirac},
28         {NULL, NULL, GHOST, -GHOST, NULL, GRCC_PT_Ghost},
29     };
30
31     /* definition of particles:
32      *   {name, code, total_ord, {p.names}, {p.codes}, {c.consts}}
33      */
34     static IInput iinp[] = {
35         {NULL, 31, 3, {NULL}, { GLUON,  GLUON, GLUON},        {1}},
36         {NULL, 32, 3, {NULL}, { GLUON, -QUARK, QUARK},        {1}},
37         {NULL, 33, 3, {NULL}, {-GHOST,  GHOST, GLUON},        {1}},
38         {NULL, 41, 4, {NULL}, { GLUON,  GLUON, GLUON, GLUON}, {2}},
39     };
40     static int npin  = sizeof(pinp)/sizeof(PInput);
41     static int niin  = sizeof(iinp)/sizeof(IInput);
42
```

```
43      /* Full generation */
44      /* q q-bar ==> gluon, g^1 */
45      static FGInput fgin = {
46          2, {NULL}, {QUARK, -QUARK},           /* initial particles */
47          1, {NULL}, {GLUON},                   /* final particles */
48          {1}                                   /* coupling constants */
49      };
50
51      /* Partial generation */
52      static PGInput pgin[] = {
53          {1, GRCC_AT_Initial, 1, 0, NULL,  QUARK},  /* initial quark */
54          {1, GRCC_AT_Initial, 1, 0, NULL, -QUARK},  /* initial anti-q */
55          {1, GRCC_AT_Final,   1, 0, NULL,  GLUON},  /* final   gluon */
56          {3, GRCC_AT_Vertex,  3, 1, NULL,      0},  /* (3-p. vertex)*3 */
57          {4, GRCC_AT_Vertex,  1, 2, NULL,      0},  /* (4-p. vertex)*1 */
58      };
59      static int npgin = sizeof(pgin)/sizeof(PGInput);
60      static int clist[GRCC_MAXNCPLG];
61
62      /* Full generation */
63      for (loop = 1; loop < 5; loop++) {
64          fgin.coupl[0] += 2;
65          fgInput(&minp, npin, pinp, niin, iinp, &fgin);
66      }
67
68      /* Partial generation */
69      clist[0] = 0;
70      for (j = 0; j < npgin; j++) {
71          clist[0] += pgin[j].cple * pgin[j].cnum;
72      }
73      pgInput(&minp, npin, pinp, niin, iinp, npgin, pgin, clist);
74
75      return 0;
76  }
```

■ Include files

```
4 #include "grccparam.h"
```

Include some definitions in `grcc`.

```
5 #include "grccfromc.h"
```

Include a header file, which includes prototypes of functions defined in `"grccfromc.cc"`:

```
void fgInput(MInput *minp, int npin, PInput *pin, int niin, IInput *iin,
             FGInput *fgin);
void pgInput(MInput *minp, int npin, PInput *pin, int niin, IInput *iin,
             int npgin, PGInput *pgin, int *clist);
```

■ Particle codes

The following macro variables represents particle codes in the model.

```
 7 #define GLUON 1
 8 #define GHOST 2
 9 #define QUARK 3
```

■ Data for definition of a model

```
20     static MInput minp = {"QCD1", GRCC_DEFBYCODE, 1, {"g"}};
```

Type `MInput` is defined in `"grccparam.h"` as

```
typedef struct {
    const char *name;       /* name of the model */
          int   defpart;    /* particle is defined by name or code */
                            /* GRCC_DEFBYNAME or GRCC_DEFBYCODE */
          int   ncouple;    /* No. of coupling constants */
    const char *cnamlist[GRCC_MAXNCPLG]; /* Names of c. constants */
} MInput;
```

The name of the model is `"QCD1"`. Macro variable `GRCC_DEFBYCODE` means that particles and interactions are identified by integer codes. The model has one coupling constant named `"g"`.

■ Data for definitions of particles

```
25     static PInput pinp[] = {
26         {NULL, NULL, GLUON,  GLUON, NULL, GRCC_PT_Vector},
27         {NULL, NULL, QUARK, -QUARK, NULL, GRCC_PT_Dirac},
28         {NULL, NULL, GHOST, -GHOST, NULL, GRCC_PT_Ghost},
29     };
```

Type `PInput` is defined in `"grccparam.h"` as

```
typedef struct {
    const char *name;        /* name of particle */
    const char *aname;       /* name of anti-particle */
          int   pcode;       /* particle code */
          int   acode;       /* anti-particle code */
    const char *ptypen;      /* type : 'GRCC_PTS_Scalar' etc */
          long  ptypec;      /* type : 'GRCC_PT_Scalar' etc */
} PInput;
```

Here three particles, gluon, quark and ghost, are defined. For quark, necessary data is

```
{NULL, NULL, QUARK, -QUARK, NULL, GRCC_PT_Dirac}
```

1. The 1st and 2nd items are name of particle and anti-particle, respectively. As this model is defined with `GRCC_DEFBYCODE`, these items can be `NULL`.
2. The 3rd and 4th items are integer codes of particle and anti-particle. They should not be 0. If these two codes are the same, this particle is considered real field.
3. The 5th item is used to define the type of the particle when the model is defined with `GRCC_DEFBYNAME`. This is not the case for this example.

4. The last item `GRCC_PT_Dirac` means that the particle is Dirac particle.

■ Data for definitions of interactions

```
34     static IInput iinp[] = {
35         {NULL, 31, 3, {NULL}, { GLUON,  GLUON, GLUON},        {1}},
36         {NULL, 32, 3, {NULL}, { GLUON, -QUARK, QUARK},        {1}},
37         {NULL, 33, 3, {NULL}, {-GHOST,  GHOST, GLUON},        {1}},
38         {NULL, 41, 4, {NULL}, { GLUON,  GLUON, GLUON, GLUON}, {2}},
39     };
```

Type `IInput` is defined in `"grccparam.h"` as

```
typedef struct {
    const char *name;          /* name of interaction */
          int   icode;         /* code of interaction */
          int   nplistn;       /* No. of particles interacting */
    const char *plistn[GRCC_MAXLEGS];     /* p. names */
          int   plistc[GRCC_MAXLEGS];     /* p. codes */
          int   cvallist[GRCC_MAXNCPLG];  /* orders of c.consts */
} IInput;
```

For example the following line defines quark-antiquark-gluon interaction:

```
{NULL, 32, 3, {NULL}, { GLUON, -QUARK, QUARK}, {1}}
```

1. The 1st item is name (character string) of the interaction. As this model is defined with `GRCC_DEFBYCODE`, this item can be `NULL`.
2. The 2nd item `32` is user defined code for this interaction. It should be positive integer (not including 0).
3. The 4th item is the list of particle names appearing in the interaction.

   Value `{NULL}` is equivalent to `{NULL, NULL, NULL, ...}` and is ignored as `GRCC_DEFBYCODE` is specified in the model.
4. The 5th item `{ GLUON, -QUARK, QUARK}` is a list of particle codes appearing in the particle definition.
5. The last item `{1}` represents the order of coupling constant is 1.

■ Data for full generation

```
45     static FGInput fgin = {
46         2, {NULL}, {QUARK, -QUARK},   /* initial particles */
47         1, {NULL}, {GLUON},           /* final particles */
48         {1}                           /* coupling constants */
49     };
```

Type `FGInput` is defined in `"grccparam.h"` as

```
typedef struct {
    long          ninitl;                 /* no. of initial p. */
    const char   *initln[GRCC_MAXNODES];  /* initial p. names */
    int           initlc[GRCC_MAXNODES];  /* initial p. codes */
    long          nfinal;                 /* no. of final p. */
    const char   *finaln[GRCC_MAXNODES];  /* final p. names */
    int           finalc[GRCC_MAXNODES];  /* final p. codes */
    int           coupl[GRCC_MAXNCPLG];   /* orders of c. consts */
} FGInput;
```

The meaning of items used in the example is as the following:

1. The 1st item '2' is the number of initial particles.
2. The 2nd item is the list of names of initial particles, which is '`{NULL}`' in this example, as `GRCC_DEFBYCODE` is specified in the model.
3. The 3rd item '`{QUARK, -QUARK}`' is the list of particles codes of initial particles. Particles are defined as in-coming.
4. The 4th item '1' is the number of final particles.
5. The 5th item '`{NULL}`' may be the list of names of final particles.
6. The 6th item '`{GLUON}`' is the list of particles codes of final particles. Particles are defined as out-going.
7. The last item '`{1}`' is the orders of coupling constants. In this model, there is only one coupling constant "`g`" and the order of coupling constant is 1 for this physical process.

■ Data for partial generation

```
52     static PGInput pgin[] = {
53         {1, GRCC_AT_Initial, 1, 0, NULL,  QUARK},  /* initial */
54         {1, GRCC_AT_Initial, 1, 0, NULL, -QUARK},  /* initial */
55         {1, GRCC_AT_Final,   1, 0, NULL,  GLUON},  /* final   */
56         {3, GRCC_AT_Vertex,  3, 1, NULL,  0},   /* (3-p.v.)*3 */
57         {4, GRCC_AT_Vertex,  1, 2, NULL,  0},   /* (4-p.v.)*1 */
58     };
```

These lines defines the topologically equivalence classes of nodes. Type `PGInput` is defined in "`grccparam.h`" as

```
typedef struct {
    int         cdeg;     /* degree of each node */
    int         ctyp;     /* type : GRCC_AT_xxx */
    int         cnum;     /* the number of nodes in the class */
    int         cple;     /* total order of c.c. of each node */
                          /* = 0 for external particle */
    const char *pname;    /* particle name defined in the model */
                          /* = NULL for vertex */
    long        pcode;    /* particle code defined in the model */
                          /* = 0 for vertex */
                          /* long = int + padding */
} PGInput;
```

For example the following data represents final gluon:

```
{1, GRCC_AT_Final,   1, 0, NULL,  GLUON}
```

1. The 1st item '1' is the number legs of a node in this class.
2. The 2nd item 'GRCC_AT_Final' means this is final particle.
3. The 3rd item '1' is the number of particles in this class.
4. The 4th item '0' is the total order of coupling constants for this node.
5. The 5th item 'NULL' may be particle name, but it is ignored in this example because of the selection of GRCC_DEFBYCODE.
6. The 6th item 'GLUON' is the particle code of gluon.

For a 4-point vertex:

```
{4, GRCC_AT_Vertex,  1, 2, NULL,      0}
```

1. The 1st item '4' means that node is 4-point vertex.
2. The 2nd item 'GRCC_AT_Vertex' means this is a vertex.
3. The 3rd item '1' is the number of nodes in this group.
4. The 4th item '2' is the total order of coupling constants.
5. The 5th and the last item 'NULL' and '0' are ignored for vertex definitions.

■ Full generation

```
63     for (loop = 1; loop < 5; loop++) {
64         fgin.coupl[0] += 2;
65         fgInput(&minp, npin, pinp, niin, iinp, &fgin);
66     }
```

Full generation is made by calling function fgInput defined in "grccfromc.cc". In this example, fgInput is called several times changing the order of coupling constants.

■ Partial generation

```
69     clist[0] = 0;
70     for (j = 0; j < npgin; j++) {
71         clist[0] += pgin[j].cple * pgin[j].cnum;
72     }
73     pgInput(&minp, npin, pinp, niin, iinp, npgin, pgin, clist);
```

Partial generation is made by calling function pgInput defined in "grccfromc.cc". Before calling this function, the list of coupling constants clist is prepared. In this case, there is only one coupling constants in this model, and it is sufficient to count the total order of coupling constants.

7.2 Interface written in C++

Program "grccfromc.cc" intermediates between "testfromc.c" and grcc.
This file is like the following:

```
 1 #include <stdlib.h>
 2 #include <string.h>
 3
 4 #include "grcc.h"
 5
 6 #ifdef GRCC_NAMESPACE
 7 using namespace Grcc;
 8 #endif
 9
10 extern "C" {
11 #include "grccfromc.h"
12 }
13
14 //============================================================

        --- user defined optional functions ---

58 //============================================================
59 Model *genmodel(MInput *minp, int npin, PInput *pin,
                     int niin, IInput *iin)
60 {
61     Model *model;
62     int    j;
63
64     model = new Model(minp);
65
66     // particles
67     for (j = 0; j < npin; j++) {
68         model->addParticle(pin+j);
69     }
70     model->addParticleEnd();
71
72     // interactions;
73     for (j = 0; j < niin; j++) {
74         model->addInteraction(iin+j);
75     }
76     model->addInteractionEnd();
77
78     return model;
79 }
80
81 //------------------------------------------------------------
82 Options *setOpt(Model *model, Counter *counter)
83 {
84     Options *opt;
85
86     // options
87     opt = new Options();

        ...
```

```
117     return opt;
118 }
119
120 //-------------------------------------------------------------
121 // Full generation
122 void fgInput(MInput *minp, int npin, PInput *pin,
                    int niin, IInput *iin, FGInput *fgin)
123 {
124     Counter  counter;
125     Options *opt;
126     Model   *model;
127     Process *proc;
128
129     int      initl[GRCC_MAXNODES], final[GRCC_MAXNODES];
130     int      pid=0, j;
131
132     /* model */
133     model = genmodel(minp, npin, pin, niin, iin);
134
135     /* options */
136     opt = setOpt(model, &counter);
137
138     // process
139     if (model->defpart == GRCC_DEFBYCODE) {
140         for (j = 0; j < fgin->ninitl; j++) {
141             initl[j] = model->findParticleCode(fgin->initlc[j]);
142         }
143         for (j = 0; j < fgin->nfinal; j++) {
144             final[j] = model->findParticleCode(fgin->finalc[j]);
145         }
146     } else {
147         for (j = 0; j < fgin->ninitl; j++) {
148             initl[j] = model->findParticleName(fgin->initln[j]);
149         }
150         for (j = 0; j < fgin->nfinal; j++) {
151             final[j] = model->findParticleName(fgin->finaln[j]);
152         }
153     }
154     proc  = new Process(pid, model, opt,
155                         fgin->ninitl, initl,
156                         fgin->nfinal, final, fgin->coupl);
157
158     // print the summary of the result
159     opt->end();
160
161     printf("mgcount = %ld, agcount = %ld\n",
162            counter.mgcount, counter.agcount);
163     fclose(counter.fp);
164
165     // delete created objects
166     delete model;
167     delete proc;
168     delete opt;
```

```
169 }
170
171 //------------------------------------------------------------
172 // Partial generation
173 void pgInput(MInput *minp, int npin, PInput *pin, int niin,
                  IInput *iin, int npgin, PGInput *pgin, int *clist)
174 {
175     Counter   counter;
176     Options  *opt;
177     Model    *model;
178     SProcess *sproc;
179
180     int cdeg[GRCC_MAXNODES], ctyp[GRCC_MAXNODES];
181     int ptcl[GRCC_MAXNODES], cnum[GRCC_MAXNODES], cple[GRCC_MAXNODES];
182     int j;
183
184     // model
185     model = genmodel(minp, npin, pin, niin, iin);
186
187     // options
188     opt = setOpt(model, &counter);
189
190     // subprocess
191     for (j = 0; j < npgin; j++) {
192         cdeg[j] = pgin[j].cdeg;
193         ctyp[j] = pgin[j].ctyp;
194         cnum[j] = pgin[j].cnum;
195         if (isATExternal(pgin[j].ctyp)) {
196             if (model->defpart == GRCC_DEFBYCODE) {
197                 ptcl[j] = model->findParticleCode(pgin[j].pcode);
198             } else {
199                 ptcl[j] = model->findParticleName(pgin[j].pname);
200             }
201             cple[j] = 0;
202         } else {  // vertex
203             ptcl[j] = 0;
204             cple[j] = pgin[j].cple;
205         }
206     }
207     sproc = new SProcess(model, NULL, opt, 0, clist, npgin,
208                          cdeg, ctyp, ptcl, cple, cnum);
209     // print sprocesses
210     sproc->prSProcess();
211
212     // generate Feynman graphs
213     sproc->generate();
214
215     // print the summary of the result
216     opt->end();
217
218     printf("mgcount = %ld, agcount = %ld\n",
219            counter.mgcount, counter.agcount);
220     fclose(counter.fp);
221
```

```
222     // delete created objects
223     delete model;
224     delete sproc;
225     delete opt;
226 }
```

■ Include files

```
  4 #include "grcc.h"
  5
  6 #ifdef GRCC_NAMESPACE
  7 using namespace Grcc;
  8 #endif
  9
 10 extern "C" {
 11 #include "grccfromc.h"
 12 }
```

Header file "grccfromc.h" is included as 'extern "C"'.

■ Definition of a model

```
 59 Model *genmodel(MInput *minp, int npin, PInput *pin,
                       int niin, IInput *iin)
```

This function genmodel defines a model as described in §5.

■ Setting up options

```
 82 Options *setOpt(Model *model, Counter *counter)
```

This function sets up the options for Feynman graph generation as same as §3.9.

■ Full generation

```
127 void fgInput(MInput *minp, int npin, PInput *pin,
                int niin, IInput *iin, FGInput *fgin)
```

This function makes full generation, which is similar to §3.10

■ Internal particle code

When the particle id is defined by user defined code with GRCC_DEFBYCODE, internal particle code of grcc is obtained by:

```
model->findParticleCode( user defined code )
```

And when the particle id is defined by user defined name with GRCC_DEFBYNAME, internal particle code of grcc is obtained by:

```
model->findParticleName(user defined name)
```

See §3.5 for detail.

■ Partial generation

```
178 void pgInput(MInput *minp, int npin, PInput *pin, int niin,
                IInput *iin, int npgin, PGInput *pgin, int *clist)
```

This function makes partial generation as same as §3.6 and §3.7.

8 Tests and performances

The number of `mgraph`s is confirmed to agree with the analytic values calculated by the combinatorial method in [4, 2]. The sum weighted by the symmetry factor is also compared with the analytic result[5]. See also `"phicount.cc"`.

For `agraph` generation the numbers of graphs weighted by the symmetry factor of Feynman graphs is confirmed to agree with the analytic values calculated in [5] for some processes in QCD with $N_f = 1$ and $N_f = 3$. And also the results are compared with `grc` for some processes in QCD with $N_f = 1$ and the standard model.

The performances are measured on a machine with `Intel(R) Core(TM) i5-2500 CPU` (2794.244MHz), and the compiler `(GCC) 4.8.5 20150623 (Red Hat 4.8.5-36)` with the same compiler options "`-g -Wall`" for both programs.

■ QCD1

The numbers of generated graphs are counted in the model `"qcd1.mdl"` (QCD with $N_f = 1$) under the conditions:

1. Only one-particle irreducible graphs (and then connected and without tadpoles).
2. Without file output, just counting the number of graphs.

```
----------------------------------------------------------------------
qcd1 (1PI, including tadpoles, without output file) 2018.12.20

  loop [coupling-constants]: Ngraphs (w.sum): grc-sec: grcc-sec

[] --> []
  2     [2]:          4 (        29/  24):      -1.00:       0.00: -1.00
  3     [4]:         19 (       221/  48):      -1.00:       0.00: -1.00
  4     [6]:        118 (       365/   9):      -1.00:       0.00: -1.00
  5     [8]:       1194 (    632027/1152):      -1.00:       0.66: -1.00
  6    [10]:      18822 (  56336077/5760):      -1.00:     411.52: -1.00

['u'] --> ['u']
  1     [2]:          1 (         1/   1):       0.00:       0.00: -1.00
  2     [4]:          7 (         6/   1):       0.00:       0.00: -1.00
  3     [6]:         97 (       899/  12):       0.00:       0.01: -1.00
  4     [8]:       1835 (      3983/   3):       0.07:       0.08: -1.00
  5    [10]:      42724 (   2117279/  72):       1.86:       1.80:  0.97
  6    [12]:    1162961 (   6909980/   9):      58.61:      46.37:  0.79

['gluon'] --> ['gluon']
  1     [2]:          4 (         3/   1):       0.00:       0.00: -1.00
  2     [4]:         23 (       197/  12):       0.01:       0.00: -1.00
  3     [6]:        324 (       228/   1):       0.00:       0.01: -1.00
  4     [8]:       6484 (    211639/  48):       0.15:       0.16:  1.07
  5    [10]:     160500 (   7548709/  72):       4.52:       3.91:  0.87
  6    [12]:    4615329 ( 557188135/ 192):     154.76:     111.63:  0.72

['u', 'u-bar'] --> ['gluon']
  1     [3]:          2 (         2/   1):       0.00:       0.00: -1.00
  2     [5]:         36 (        63/   2):       0.00:       0.00: -1.00
```

```
  3     [7]:        798 (       643/   1):       0.03:       0.05: -1.00
  4     [9]:      20991 (    380275/  24):       0.84:       1.10:  1.31
  5    [11]:     628718 (   4048936/   9):      27.81:      29.99:  1.08
  6    [13]:   20967637 (1376320979/  96):    1086.63:     906.18:  0.83

['gluon', 'gluon'] --> ['gluon']
  1     [3]:          8 (        13/   2):       0.00:       0.00: -1.00
  2     [5]:        112 (       377/   4):       0.00:       0.00: -1.00
  3     [7]:       2700 (     16657/   8):       0.05:       0.07: -1.00
  4     [9]:      76261 (   2651201/  48):       1.57:       1.98:  1.26
  5    [11]:    2433302 (  13358987/   8):      56.09:      60.82:  1.08
  6    [13]:   85607767 (10771504601/192):    2243.61:    2103.30:  0.94

['u', 'u'] --> ['u', 'u']
  1     [4]:          4 (         4/   1):       0.00:       0.00: -1.00
  2     [6]:        106 (        98/   1):       0.01:       0.02: -1.00
  3     [8]:       3118 (      8023/   3):       0.36:       0.56:  1.56
  4    [10]:     100894 (    487531/   6):      11.48:      15.64:  1.36
  5    [12]:    3560900 (  16302779/   6):     427.65:     466.64:  1.09

['u', 'gluon'] --> ['u', 'gluon']
  1     [4]:          7 (         7/   1):       0.00:       0.00: -1.00
  2     [6]:        235 (       421/   2):       0.02:       0.03: -1.00
  3     [8]:       7737 (     77101/  12):       0.39:       0.63:  1.62
  4    [10]:     271248 (   5071721/  24):      13.51:      18.71:  1.38
  5    [12]:   10183532 (  67614674/   9):     521.82:     597.04:  1.14

['gluon', 'gluon'] --> ['gluon', 'gluon']
  1     [4]:         24 (        45/   2):       0.00:       0.00: -1.00
  2     [6]:        751 (      2653/   4):       0.02:       0.03: -1.00
  3     [8]:      26991 (    174361/   8):       0.60:       0.91:  1.52
  4    [10]:    1017480 (  12270107/  16):      22.78:      30.52:  1.34
  5    [12]:   40595008 ( 231078281/   8):     965.22:    1118.90:  1.16

['u', 'u'] --> ['u', 'u', 'gluon']
  1     [5]:         16 (        16/   1):       0.00:       0.00: -1.00
  2     [7]:        772 (       720/   1):       0.15:       0.30:  2.00
  3     [9]:      32772 (     85766/   3):       5.59:       9.87:  1.77

['u', 'gluon'] --> ['u', 'gluon', 'gluon']
  1     [5]:         33 (        33/   1):       0.00:       0.01: -1.00
  2     [7]:       1848 (      3375/   2):       0.17:       0.32:  1.88
  3     [9]:      85780 (    291765/   4):       6.14:      10.76:  1.75
  4    [11]:    3890270 (  24865821/   8):     246.74:     375.03:  1.52

['gluon', 'gluon'] --> ['gluon', 'gluon', 'gluon']
  1     [5]:        105 (       105/   1):       0.01:       0.01: -1.00
  2     [7]:       6100 (      5530/   1):       0.20:       0.38:  1.90
  3     [9]:     309230 (    257220/   1):       8.62:      14.11:  1.64
  4    [11]:   15046760 (  11684875/   1):     390.70:     555.19:  1.42

['u', 'u'] --> ['u', 'u', 'u', 'u-bar']
  1     [6]:         48 (        48/   1):       0.06:       0.14: -1.00
  2     [8]:       3192 (      3048/   1):       2.22:       5.43:  2.45
```

```
  3    [10]:     170088 (    153480/   1):      95.82:     206.16:  2.15
  4    [12]:    8589654 (   7340406/   1):    4261.99:    7683.40:  1.80

['u', 'u'] --> ['u', 'u', 'gluon', 'gluon']
  1     [6]:         88 (        88/   1):       0.06:       0.14: -1.00
  2     [8]:       6716 (      6320/   1):       2.25:       5.49:  2.44
  3    [10]:     390970 (    345946/   1):      98.54:     209.81:  2.13
  4    [12]:   21082840 (  52856978/   3):    4443.58:    7933.41:  1.79

['u', 'gluon'] --> ['u', 'gluon', 'gluon', 'gluon']
  1     [6]:        198 (       198/   1):       0.05:       0.14: -1.00
  2     [8]:      16998 (     15750/   1):       2.36:       5.72:  2.42
  3    [10]:    1068390 (    925861/   1):     106.44:     222.43:  2.09
  4    [12]:   61204923 (  99859609/   2):    4985.95:    8734.56:  1.75

['gluon', 'gluon'] --> ['gluon', 'gluon', 'gluon', 'gluon']
  1     [6]:        630 (       630/   1):       0.05:       0.15: -1.00
  2     [8]:      58035 (     53415/   1):       2.81:       6.31:  2.25
  3    [10]:    3975690 (   3380055/   1):     141.56:     270.09:  1.91
  4    [12]:  243734380 ( 193845535/   1):    7496.51:   11870.01:  1.58
-------------------------------------------------------------------------
```

■ Standard model

The numbers of generated graphs counted in the model `"standard.mdl"` under the same conditions with the case of `"qcd1.mdl"`.

The numbers of graphs are calculated in [5] only for the case of electroweak coupling constant being zero, where the model is equivalent to QCD with $N_f = 6$. The numbers of these processes are compared with analytic values.

```
-------------------------------------------------------------------------
standard (1PI, including tadpoles, without output file) 2018.12.20

  loop [coupling-constants]: Ngraphs (w.sum): grc-sec: grcc-sec

[] --> []
  2  [2, 0]:         90 (       643/  12):      -1.00:       0.00: -1.00
  3  [4, 0]:       3218 (    115357/  48):      -1.00:       0.17: -1.00
  4  [6, 0]:     277561 (  72061801/ 288):      -1.00:      17.48: -1.00
  2  [0, 2]:          9 (        89/  24):      -1.00:       0.00: -1.00
  3  [2, 2]:        150 (       126/   1):      -1.00:       0.15: -1.00
  4  [4, 2]:      14790 (     27975/   2):      -1.00:      16.21: -1.00
  3  [0, 4]:         69 (      1141/  48):      -1.00:       0.01: -1.00
  4  [0, 6]:        628 (     11465/  36):      -1.00:       0.06: -1.00
  5  [0, 8]:       9889 (   7313267/1152):      -1.00:       1.49: -1.00

['u'] --> ['u']
  1  [2, 0]:          6 (         6/   1):       0.00:       0.00: -1.00
  2  [4, 0]:        459 (       899/   2):       0.01:       0.01: -1.00
  3  [6, 0]:      65508 (    190096/   3):       0.98:       1.28:  1.31
  1  [0, 2]:          1 (         1/   1):       0.00:       0.00: -1.00
  2  [2, 2]:         72 (        72/   1):       0.00:       0.01: -1.00
  3  [4, 2]:      10330 (     10226/   1):       0.78:       1.22:  1.56
  2  [0, 4]:         12 (        11/   1):       0.00:       0.00: -1.00
  3  [0, 6]:        242 (      2489/  12):       0.53:       0.02:  0.04
```

```
  4  [0, 8]:       6525 (     64357/  12):     131.23:       0.29:  0.00

['gluon'] --> ['gluon']
  1  [0, 2]:          9 (         8/   1):       0.00:       0.00: -1.00
  2  [0, 4]:         58 (       587/  12):       0.00:       0.00: -1.00
  3  [0, 6]:       1319 (      4457/   4):       0.22:       0.05:  0.23
  4  [0, 8]:      39924 (    523703/  16):      43.33:       1.31:  0.03

['z'] --> ['z']
  1  [2, 0]:         24 (        23/   1):       0.01:       0.00: -1.00
  2  [4, 0]:       1156 (      4407/   4):       0.01:       0.02: -1.00
  3  [6, 0]:     197519 (   4550575/  24):       3.54:       3.65:  1.03
  1  [0, 2]:          0 (         0/   1):       0.00:       0.00: -1.00
  2  [2, 2]:         18 (        18/   1):       0.01:       0.02: -1.00
  3  [4, 2]:       4224 (      4212/   1):       2.73:       2.93:  1.07

['higgs'] --> ['higgs']
  1  [2, 0]:         25 (        22/   1):       0.00:       0.00: -1.00
  2  [4, 0]:       1255 (     14033/  12):       0.02:       0.02: -1.00
  3  [6, 0]:     211101 (   4829851/  24):       3.76:       3.96:  1.05
  1  [0, 2]:          0 (         0/   1):       0.00:       0.00: -1.00
  2  [2, 2]:         18 (        18/   1):       0.01:       0.02: -1.00
  3  [4, 2]:       4635 (      4617/   1):       2.90:       3.16:  1.09

['u', 'u-bar'] --> ['gluon']
  1  [0, 3]:          2 (         2/   1):       0.00:       0.00: -1.00
  2  [0, 5]:         61 (       113/   2):       0.01:       0.00: -1.00
  3  [0, 7]:       1978 (      3481/   2):       1.80:       0.11:  0.06

['u', 'u-bar'] --> ['z']
  1  [3, 0]:         14 (        14/   1):       0.00:       0.00: -1.00
  2  [5, 0]:       2800 (      5511/   2):       0.03:       0.04: -1.00
  3  [7, 0]:     675578 (   1972213/   3):       8.77:      11.17:  1.27
  1  [1, 2]:          1 (         1/   1):       0.00:       0.00: -1.00
  2  [3, 2]:        296 (       296/   1):       0.02:       0.04: -1.00
  3  [5, 2]:      79868 (     79188/   1):       7.27:       9.70:  1.33

['higgs', 'higgs'] --> ['z']
  1  [3, 0]:         44 (        44/   1):       0.00:       0.00: -1.00
  2  [5, 0]:       8174 (      8027/   1):       0.10:       0.12: -1.00
  3  [7, 0]:    2451053 (   2373796/   1):      34.68:      38.23:  1.10
  1  [1, 2]:          0 (         0/   1):       0.00:       0.00: -1.00
  2  [3, 2]:         72 (        72/   1):       0.08:       0.09: -1.00
  3  [5, 2]:      35208 (     35208/   1):      28.20:      28.47:  1.01

['u', 'u'] --> ['u', 'u']
  1  [0, 4]:          4 (         4/   1):       0.01:       0.00: -1.00
  2  [0, 6]:        146 (       138/   1):       0.13:       0.03:  0.23
  3  [0, 8]:       5838 (     15763/   3):      38.54:       0.88:  0.02
  4 [0, 10]:     255954 (   1327051/   6):   13666.59:      27.52:  0.00

['u', 'gluon'] --> ['u', 'gluon']
  1  [0, 4]:          7 (         7/   1):       0.00:       0.00: -1.00
  2  [0, 6]:        395 (       741/   2):       0.06:       0.04: -1.00
```

```
  3  [0, 8]:      19027 (    204421/  12):      10.55:       1.21:  0.11

['gluon', 'gluon'] --> ['gluon', 'gluon']
  1  [0, 4]:         54 (       105/   2):       0.00:       0.00: -1.00
  2  [0, 6]:       1966 (      7513/   4):       0.05:       0.07: -1.00
  3  [0, 8]:     111551 (    803681/   8):       7.72:       3.30:  0.43

['u', 'photon'] --> ['u', 'photon']
  1  [2, 2]:          2 (         2/   1):       0.00:       0.00: -1.00
  2  [2, 4]:        108 (       106/   1):       0.12:       0.09:  0.75
  3  [2, 6]:       4638 (     26345/   6):      40.20:       6.45:  0.16
--------------------------------------------------------------------------
```

References

[1] T. Kaneko, *Comput. Phys. Commun.* **92** (1995) 127–152.

Program `grc.v2.2` (including `grc`, `grcdraw` and `grcplot`) is available from:

`http://research.kek.jp/people/kaneko/`

[2] P. Nogueira, *J. Comput. Phys.* **105** (1993) 279.

[3] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "*The Design and Analysis of Computer Algorithms*", Addison-Wesley, 1974, "§5.3 Biconnectivity".

[4] F. Harary and E. P. Palmer, "*Graphical Enumeration*", Academic Press, 1973.

[5] T. Kaneko, *Comput. Phys. Commun.* **226** (2108) 104–113.

Some results are found in the same URL as in [1].