Published by Institute of Physics Publishing for SISSA/ISAS
Received: *September 06, 2001*
Revised: *January 23, 2002*
Accepted: *February 27, 2002*

AMEGIC++ 1.0
A Matrix Element Generator In C++

Frank Krauss
Cavendish Laboratory, University of Cambridge
Cambridge CB3 0HE, UK
E-mail: krauss@hep.phy.cam.ac.uk

Ralf Kuhn* **and Gerhard Soff**
Institut für Theoretische Physik, TU Dresden
01062 Dresden, Germany
E-mail: kuhn@theory.phy.tu-dresden.de, soff@physik.tu-dresden.de

Abstract: The new matrix element generator AMEGIC++ is introduced, dedicated to describe multi-particle production in high energy particle collisions. It automatically generates helicity amplitudes for the processes under consideration and constructs suitable, efficient integration channels for the multi-channel phase space integration. The corresponding expressions for the amplitudes and the integrators are stored in library files to be linked to the main program.

Keywords: Standard Model, Higgs Physics, LEP HERA and SLC Physics, QCD.

*Max Planck Institut für Physik Komplexer Systeme, 01187 Dresden, Germany.

JHEP02(2002)044

Contents

Program summary

Title of the program: `AMEGIC++`

Program obtainable from: authors upon request

Licensing provisions: none

Operating systems under which the program has been tested: UNIX, LINUX, VMS

Programming language: C++

Separate documentation available: no

Nature of the physical problem: The theoretical description of multi particle production, even at the tree-level, suffers from two problems:

1. The rapidly increasing number of amplitudes forbids the traditional method of summing and squaring individual Feynman amplitudes by means of the completeness relations for spinors and polarization vectors. Instead, the helicity method is employed, translating the amplitudes into complex numbers. Still, the helicity amplitudes for a large number of diagrams have to be constructed which itself is a formidable task.

2. The complex structure of the high-dimensional phase space imperatively requires using Monte Carlo methods for its evaluation. Here, efficiency is of paramount importance, and one has to employ non-flat phase space measures which must be optimized for the process under consideration and its specific singularity structure in phase space.

Method of solution: Automatic generation of helicity amplitudes related to Feynman diagrams describing the process at the tree-level. Translating of the amplitudes into character strings and storing in libraries for increased efficiency during evaluation, i.e. phase space integration. Integration by means of multichannel methods with specific channels which are constructed from the Feynman diagrams.

1. Introduction

In the last decades, the observation of the production and subsequent decay of particles with rising masses [1] has lead to more and more far reaching qualitative and increasingly accurate quantitative knowledge about physics on subnuclear scales. As a consequence of this experimental success, the Standard Model [2] nowadays is widely accepted as the best model for the scales under current consideration. However, many particle physicists believe that the Standard Model is only the effective realization of a theory valid up to much higher scales with even higher energetic particles participating. Consequently, for the experimental verification or falsification of such ideas, colliders with even higher energy release are currently being planned or constructed [3]. From a theoretical point of view the increasing energy release does not only allow the creation of these presumably unstable new particles, in fact, with rising energies the number of particles or jets produced increases drastically as well. Therefore, to analyse both the signal and the background related to processes, where new heavy particles are produced and decay into high energetic secondaries, the description of multi-particle states is mandatory.

However, such a description, even in the lowest accessible perturbative approximation is far from being trivial. Basically, two major problems emerge with an increasing number of final state particles:

1. The number of amplitudes related to a process with n outgoing particles rises approximately like a factorial in n. Thus, the traditional method of summing and squaring the amplitudes by use of the completeness relations leads to an exploding number of terms to be evaluated.

2. The phase space populated by the final state particles is multidimensional, enforcing the use of Monte Carlo methods for the integration. However, non-trivial experimental cuts on the phase space in connection with possibly wildly fluctuating integrands due to nearly on-shell intermediate particles make sophisticated mapping and sampling methods indispensable.

These two problems necessitate the usage of computer programs. Examples are `CompHep` [4], `FeynArts/FeynCalc` [5], `Grace` [6], `MADGRAPH` [7], and `O'Mega/WHIZARD` [8].

The new program `AMEGIC++` which will be presented in this publication, proposes a solution to both problems discussed above. In its present form, `AMEGIC++` automatically creates all Feynman diagrams related to user defined processes in the framework of the Standard Model. It should be emphasized here that a generalization to a large variety of other models is straightforward and currently in preparation. However, the diagrams are then transformed into helicity amplitudes, yielding complex numbers for every combination of external momenta. These amplitudes, corresponding to a specific process, are translated into character strings and stored in a library. For the phase space integration, the structure of the individual Feynman diagrams is analysed and suitable integration channels are generated. Again, they are translated into a library file. In this sense, `AMEGIC++` is a program to generate highly efficient matrix element generators for a large variety of processes. Of course, this allows an easy implementation into Monte Carlo event generators like `APACIC++` (for a detailed description see [9]).

The outline of this paper is as follows: first, the theoretical background for both the evaluation of amplitudes and the efficient Monte Carlo generation of phase space points is reviewed, see section 2. Special emphasis is given to the helicity method, section 2.1, and on some of the features of this method as implemented within `AMEGIC++`. Additionally, the multi-channel algorithm for an efficient integration is discussed in some detail and both default channels available in `AMEGIC++` and building blocks for process dependent channels are listed. At the end, some higher order effects will be discussed in section 2.3.

Then, large parts of the algorithms involved are discussed, see section 3. They can roughly be divided into three parts, namely into methods for the generation of diagrams presented in section 3.1, for their translation into helicity amplitudes, cf. section 3.2, and finally, into algorithms for the construction of suitable integration channels, see section 3.3.

In section 4 the corresponding classes are listed and their interplay is described. This description is roughly split into the following parts:

- First, section 4.1 deals with the overhead, i.e. classes to drive `AMEGIC++`.

- Then, in section 4.2, the general set-up of models is discussed.
- Section 4.3 describes the classes responsible for the construction of Feynman diagrams, their translation into helicity amplitudes and their calculation.
- In section 4.4 the essential methods for the translation of the helicity amplitudes into character strings and their storage in corresponding library files will be highlighted.
- Section 4.5 explains the construction of the phase space integrators, the channels, and how they are used during the integration.
- In section 4.6, the parameters and switches to be set by the user are explained and how they are read in.
- Finally, in section 4.7 some more general helper classes are listed.

At the end an installation guide is given in section 5 and a summary, section 6 concludes this paper. Note that a Test Run Output can be found in appendix D as well.

2. Theoretical background

Before we present the algorithms and their implementation in `AMEGIC++` in some detail, we will give a brief review of the underlying calculational methods. More specifically, in section 2.1 we will dwell on the method of helicity amplitudes as introduced in [10, 11] and clarify our notation. Then, in section 2.2, we will review the multi-channel methods [12] employed for the subsequent phase space integration of the matrix element.

2.1 Helicity amplitudes

The basic idea behind the method of helicity amplitudes is that for every combination of helicities and momenta for the incoming and outgoing particles any Feynman amplitude (represented by propagators and vertices for the internal lines and spinors and polarization vectors for the external particles) is nothing else but a complex number. Once every such number is computed, all amplitudes for a process under consideration can be easily summed and the result can then be squared. Inclusion of colour factors etc. is straightforward and reduces in this framework to the multiplication of two vectors of complex numbers (for the amplitudes and their complex conjugated) with a colour matrix in the middle. The question to be answered in this section is how we can extract the numerical value of the amplitudes as a function of the external momenta and helicities. In `AMEGIC++`, we follow quite an old approach to answer this question which has been proven to be adequate and successful in a large number of specific processes [11].[1] This approach relies on decomposition of

[1]However, we want to point out that other approaches exist [13] which are based on the LSZ theorem and allow the direct recursive construction of matrix elements without taking the detour of Feynman amplitudes. These approaches are currently under further investigation and discussion. Nevertheless, we are convinced that in the light of the need for efficient mappings of the phase space to specific channels reproducing the singularity structure of the process under consideration, the construction of Feynman amplitudes is still a sine qua non.

the amplitude into scalar products of four-momenta and into spinor products of the form $u(p_i,\lambda_i)\bar{u}(p_j,\lambda_j)$, where the $p_{i,j}$ and $\lambda_{i,j}$ are momenta and helicities, respectively. Knowing the four-momenta, the scalar products can be calculated easily, and for fixed helicities, the spinor products can be computed as well. As we will see, the result for such a spinor product is a function of the components of the momenta involved. This allows for the translation of any Feynman amplitude into generic building blocks which can be evaluated numerically. In the following we will give a short introduction into the spinor techniques we use for this decomposition and we will list the building blocks employed in `AMEGIC++`.

2.1.1 Spinors

As it is well known, appropriate spinors $u(p,\lambda)$ for fermions with mass m, momentum p and with definite helicity λ obey Dirac's equation of motion (E.o.M.)

$$(\not{p}-m)u(p,\lambda)=0\,,\qquad (\not{p}+m)v(p,\lambda)=0\,. \tag{2.1}$$

Note that since $\not{p}$ is not necessarily hermitian, m does not need to be real, but in any case $p^2=m^2$ is fulfilled. With $p^2<0$, m turns imaginary and one can identify the particle spinor with the positive imaginary eigenvalue m ($\mathrm{Im}(m)>0$). Additionally, the spinors should fulfill

$$(1\mp\gamma^5\not{s})u(p,\pm)=0\,,\qquad (1\mp\gamma^5\not{s})v(p,\pm)=0 \tag{2.2}$$

with the polarization vector s obeying $s\cdot p=0$ and $s^2=1$.

To construct spinors satisfying both eqs. (2.1) and (2.2), even for imaginary masses, we start with massless chiral spinors w, i.e. spinors satisfying

$$w(k_0,\lambda)\bar{w}(k_0,\lambda)=\frac{1+\lambda\gamma_5}{2}\not{k}_0\,, \tag{2.3}$$

for arbitrary light-like k_0. Starting from such a chiral spinor $w(k_0,\lambda)$ with definite chirality, spinors of opposite chirality $w(k_0,-\lambda)$ can be constructed via

$$w(k_0,\lambda)=\lambda\not{k}_1 w(k_0,-\lambda) \tag{2.4}$$

with the vectors $k_{0,1}$ satisfying[2]

$$k_0^2=0\,,\qquad k_0\cdot k_1=0\,,\qquad k_1^2=-1\,. \tag{2.5}$$

Massive spinors can then be expressed as

$$u(p,\lambda)=\frac{\not{p}+m}{\sqrt{2p\cdot k_0}}\,w(k_0,-\lambda)\,,\qquad v(p,\lambda)=\frac{\not{p}-m}{\sqrt{2p\cdot k_0}}\,w(k_0,-\lambda) \tag{2.6}$$

in terms of these chiral spinors. These relations hold also true for $p^2<0$ and imaginary m. For the construction of conjugate spinors we will abandon the proper definition

$$\bar{u}=u^\dagger\gamma^0\,,\qquad \bar{v}=v^\dagger\gamma^0\,, \tag{2.7}$$

[2]Note that we have some freedom in choosing the k-vectors which provides a valuable tool for checking the calculation.

since this definition does not lead to the E.o.M.

$$\bar{u}(p,\lambda)(\not{p}-m)=0\,,\qquad \bar{v}(p,\lambda)(\not{p}+m)=0 \tag{2.8}$$

for spinors with imaginary masses. Instead we use eqs. (2.8) and

$$\bar{u}(p,\pm)(1\mp\gamma^5\not{s})=0\,,\qquad \bar{v}(p,\pm)(1\mp\gamma^5\not{s})=0 \tag{2.9}$$

to define conjugate spinors. With this definition, and choosing the normalization

$$\bar{u}(p,\lambda)u(p,\lambda)=2m\,,\qquad \bar{v}(p,\lambda)v(p,\lambda)=-2m\,, \tag{2.10}$$

the conjugate spinors are constructed as

$$\bar{u}(p,\lambda)=\bar{w}(k_0,-\lambda)\,\frac{\not{p}+m}{\sqrt{2p\cdot k_0}}\,,\qquad \bar{v}(p,\lambda)=\bar{w}(k_0,-\lambda)\,\frac{\not{p}-m}{\sqrt{2p\cdot k_0}}\,. \tag{2.11}$$

They fulfill the completeness relation

$$1=\sum_\lambda \frac{\bar{u}(p,\lambda)u(p,\lambda)-\bar{v}(p,\lambda)v(p,\lambda)}{2m}\,. \tag{2.12}$$

The last relation can now be used to decompose the numerators of fermion propagators according to

$$\begin{aligned}
&\bar{u}(p_1,\lambda_1)\,\chi_1\,(\not{p}_2+\mu_2)\,\chi_3\,u(p_3,\lambda_3)\longrightarrow\\
&\longrightarrow\frac{1}{2}\left[\sum_{\lambda_2}\bar{u}(p_1,\lambda_1)\,\chi_1\,u(p_2,\lambda_2)\,\bar{u}(p_2,\lambda_2)\,\chi_3\,u(p_3,\lambda_3)\,\left(1+\frac{\mu_2}{m_2}\right)+\right.\\
&\qquad\left.+\,\bar{u}(p_1,\lambda_1)\,\chi_1\,v(p_2,\lambda_2)\,\bar{v}(p_2,\lambda_2)\,\chi_3\,u(p_3,\lambda_3)\,\left(1-\frac{\mu_2}{m_2}\right)\right]\,,
\end{aligned} \tag{2.13}$$

where $m_2^2=p_2^2$ and the χ denote the (scalar or vector) couplings involved.

2.1.2 Polarization vectors

1. **Massless bosons:** The next task on our list is to translate the polarization vectors $\epsilon(p,\lambda)$ of massless external spin-1 bosons with momentum p and helicity λ into suitable spinor products. Massive bosons will experience a different treatment, we will deal with them later. However, for massless bosons, the polarization vectors satisfy

$$\begin{aligned}
\epsilon(p,\lambda)\cdot p&=0\,, & \epsilon(p,\lambda)\cdot\epsilon(p,\lambda)&=0\,,\\
\epsilon^\mu(p,-\lambda)&=\epsilon^{\mu*}(p,\lambda)\,, & \epsilon(p,\lambda)\cdot\epsilon(p,-\lambda)&=-1\,.
\end{aligned} \tag{2.14}$$

   In the axial gauge the polarization sum reads

$$\sum_{\lambda=\pm}\epsilon^\mu(p,\lambda)\epsilon^{\nu*}(p,\lambda)=-g^{\mu\nu}+\frac{q^\mu p^\nu+q^\nu p^\mu}{p\cdot q}\,, \tag{2.15}$$

where q^μ is some arbitrary, light-like four vector not parallel to p^μ. It can be shown that the spinor objects

$$\frac{1}{\sqrt{4p \cdot q}}\, \bar{u}(q,\lambda)\gamma^\mu u(p,\lambda) \tag{2.16}$$

satisfy the properties of eqs. (2.14) and (2.15) and hence form an acceptable choice for the polarization vectors of massless external spin-1 bosons. From this identification, we obtain an additional constraint for q^μ, namely that it should not be collinear to k_0^μ. Note, however, that the freedom in choosing q reflects the freedom in choosing a gauge vector in the axial gauge. Since final results should not depend on such a choice, we have another tool at hand to check our calculations.

2. **Massive bosons:** For massive bosons, the situation changes. First of all, there is another, longitudinal polarization state which satisfies the same properties as the transversal states, eq. (2.14). In contrast to the massless case, however, the spin sum changes and becomes

$$\sum_{\lambda=\pm,0} \epsilon^\mu(p,\lambda)\epsilon^{\nu *}(p,\lambda) = -g^{\mu\nu} + \frac{p^\mu p^\nu}{m^2}\,, \tag{2.17}$$

where m is the mass of the boson. We introduce

$$a^\mu = \bar{u}(r_2,-)\gamma^\mu u(r_1,-) \tag{2.18}$$

with light-like four vectors r_1 and r_2 satisfying

$$r_1^2 = r_2^2 = 0\,, \qquad \text{and} \qquad r_1^\mu + r_2^\mu = p^\mu\,. \tag{2.19}$$

Replacing the spin sum by an integration over the solid angle of r_1 in the rest frame of p, we obtain

$$\int_{4\pi} d\Omega\, a^\mu a^{\nu *} = \frac{8\pi m^2}{3}\left(-g^{\mu\nu} + \frac{p^\mu p^\nu}{m^2}\right) \tag{2.20}$$

which is of the desired form. Therefore, identifying

$$\epsilon^\mu \Longrightarrow a^\mu\,, \qquad \text{and} \qquad \sum_{\lambda=\pm,0} \Longrightarrow \frac{3}{8\pi m^2}\int_{4\pi} d\Omega \tag{2.21}$$

we will arrive at the correct form for the unpolarized cross section.

In other words, the polarization vector of a massive vector boson can be constructed in terms of spinors via a pseudo-decay of the boson into two massless fermions with coupling constant 1.

2.1.3 Building blocks

Finally, we will present the basic spinor products which serve as elementary building blocks for our amplitudes. We introduce the following short-hand notations:

- projectors

$$P_{L,R} = \frac{1 \mp \gamma_5}{2}\,; \tag{2.22}$$

$\lambda_1\lambda_2$	$Y(p_1,\lambda_1;p_2,\lambda_2;c_L,c_R)$	$\lambda_1\lambda_2$	$Y(p_1,\lambda_1;p_2,\lambda_2;c_L,c_R)$
$++$	$c_R\mu_1\eta_2 + c_L\mu_2\eta_1$	$+-$	$c_L S(+;p_1,p_2)$

Table 1: Y-functions for different helicity combinations (λ_1, λ_2). The remaining Y-functions can be obtained by exchanging $L \leftrightarrow R$ and $+ \leftrightarrow -$.

$\lambda_1\lambda_3$	$X(p_1,\lambda_1;p_2;p_3,\lambda_3;c_L,c_R)$
$++$	$(\mu_1\eta_2+\mu_2\eta_1)\,(\mu_2\eta_3 c_R + \mu_3\eta_2 c_L) + c_R S(+;p_1,p_2)S(-;p_2,p_3)$
$+-$	$c_L\,(\mu_1\eta_2+\mu_2\eta_1)\,S(+;p_2,p_3) + (c_L\mu_2\eta_3 + c_R\mu_3\eta_2)\,S(+;p_1,p_2)$

Table 2: X-functions for different helicity combinations (λ_1, λ_3). The remaining X-functions can be obtained by exchanging $L \leftrightarrow R$ and $+ \leftrightarrow -$.

- S-functions

$$S(+;p_1,p_2) = 2c_L \frac{k_0\cdot p_1 k_1\cdot p_2 - k_0\cdot p_2 k_1\cdot p_1 + i\epsilon_{\mu\nu\rho\sigma}k_0^\mu k_1^\nu p_1^\rho p_2^\sigma}{\eta_1\eta_2} \tag{2.23}$$

  with the symmetry properties

$$S(-;p_1,p_2) = S^*(+;p_2,p_1)\,, \qquad \text{and} \qquad S(\pm;p_1,p_2) = -S(\pm;p_2,p_1)\,; \tag{2.24}$$

- normalizations and mass-terms

$$\eta_i = \sqrt{2p_i\cdot k_0}\,, \qquad \text{and} \qquad \mu_i = \pm\frac{m_i}{\eta_i}\,, \tag{2.25}$$

  where the signs denote particles and anti-particles.

The following spinor expressions are then the basic building blocks for helicity amplitudes within `AMEGIC++` :

1. $Y(p_1,\lambda_1;p_2,\lambda_2;c_L,c_R) = \bar u(p_1,\lambda_1)[c_L P_L + c_R P_R]u(p_2,\lambda_2)\,;$
2. $X(p_1,\lambda_1;p_2;p_3,\lambda_3;c_L,c_R) = \bar u(p_1,\lambda_1)\not p_2[c_L P_L + c_R P_R]u(p_3,\lambda_3)\,;$
3. $Z(p_1,\lambda_1;p_2,\lambda_2;p_3,\lambda_3;p_4,\lambda_4;c_L^{12},c_R^{12};c_L^{34},c_R^{34}) =$
   $\bar u(p_1,\lambda_1)\gamma^\mu[c_L^{12}P_L + c_R^{12}P_R]u(p_2,\lambda_2)\bar u(p_3,\lambda_3)\gamma_\mu[c_L^{34}P_L + c_R^{34}P_R]u(p_4,\lambda_4)\,.$

Their explicit form can be found in tables 1, 2, and 3. However, let us note that, in general, products like

$$\bar u(p_1)\gamma_\mu u(p_2)\bar u(p_3)\gamma_\nu u(p_4)\cdot\left(g^{\mu\nu} - \frac{P^\mu P^\nu}{M^2}\right) \tag{2.26}$$

(stemming from the exchange of a massive boson with mass M and momentum P between the two fermion lines) cannot be represented in terms of one elementary Z-function per helicity combination. Instead, some composite functions, consisting of Z- and X-functions, become necessary.

$\lambda_1\lambda_2\lambda_3\lambda_4$	$Z(p_1,\lambda_1;p_2,\lambda_2;p_3,\lambda_3;p_4,\lambda_4;c_L^{12},c_R^{12};c_L^{34},c_R^{34})$
$++++$	$2\left[S(+;p_3,p_1)S(-;p_2,p_4)c_R^{12}c_R^{34}+\mu_1\mu_2\eta_3\eta_4c_L^{12}c_R^{34}+\mu_3\mu_4\eta_1\eta_2c_R^{12}c_L^{34}\right]$
$+++-$	$2\eta_2c_R^{12}\left[S(+;p_1,p_4)\mu_3c_L^{34}+S(+;p_1,p_3)\mu_4c_R^{34}\right]$
$++-+$	$2\eta_1c_R^{12}\left[S(-;p_3,p_2)\mu_4c_L^{34}q+S(-;p_4,p_2)\mu_3c_R^{34}\right]$
$++--$	$2\left[S(+;p_4,p_1)S(-;p_2,p_3)c_R^{12}c_L^{34}+\mu_1\mu_2\eta_3\eta_4c_L^{12}c_L^{34}+\mu_3\mu_4\eta_1\eta_2c_R^{12}c_R^{34}\right]$
$+-++$	$2\eta_4c_R^{34}\left[S(+;p_1,p_3)\mu_2c_R^{12}+S(+;p_2,p_3)\mu_1c_L^{12}\right]$
$+-+-$	0
$+--+$	$-2\left[\mu_1\mu_4\eta_2\eta_3c_L^{12}c_L^{34}+\mu_2\mu_3\eta_1\eta_4c_R^{12}c_R^{34}-\mu_1\mu_3\eta_2\eta_4c_L^{12}c_R^{34}-\mu_2\mu_4\eta_1\eta_3c_R^{12}c_L^{34}\right]$
$+---$	$2\eta_3c_L^{34}\left[S(+;p_4,p_2)\mu_1c_L^{12}+S(+;p_1,p_4)\mu_2c_R^{12}\right]$

Table 3: Z-functions for different helicity combinations (λ_1, λ_2, λ_3, λ_4). The remaining Z-functions can be obtained by exchanging $L \leftrightarrow R$ and $+ \leftrightarrow -$.

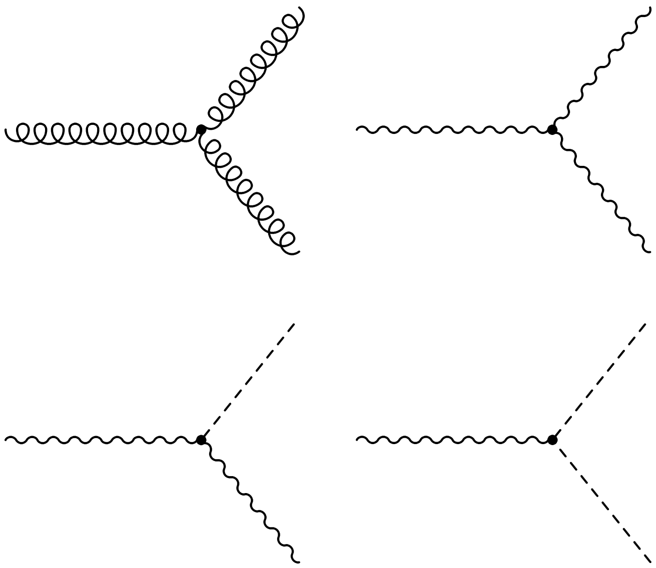


Figure 1: Three boson vertices available within `AMEGIC++`. Note that all boson lines might be on- or off-shell. Scalar and vector bosons are depicted as dotted and wavy lines, respectively, whereas a gluon is marked by a curly line.

Similarly, the propagation of longitudinal modes in off-shell vector bosons prevents us from employing polarization vectors to replace the numerators of the propagators. Therefore, structures with three or more bosons can not be "cut open" and suitable building blocks need to be constructed for any such structure. Within `AMEGIC++`, building blocks for the structures depicted in figures 1, 2, and 3 are available at the moment, somewhat limiting the maximal number of vector boson lines. Note that since vertices involving only scalar bosons are proportional to scalars, there is no need to define specific three or four scalar vertices. Similarly, scalar propagators can easily be cut open and even more complicated structures can be decomposed into easier ones as depicted in figure 4. Here, we feel, a further clarification is in order. As the observant reader might have noticed, we omitted vertices with four vector bosons in our pictorial presentation of the building

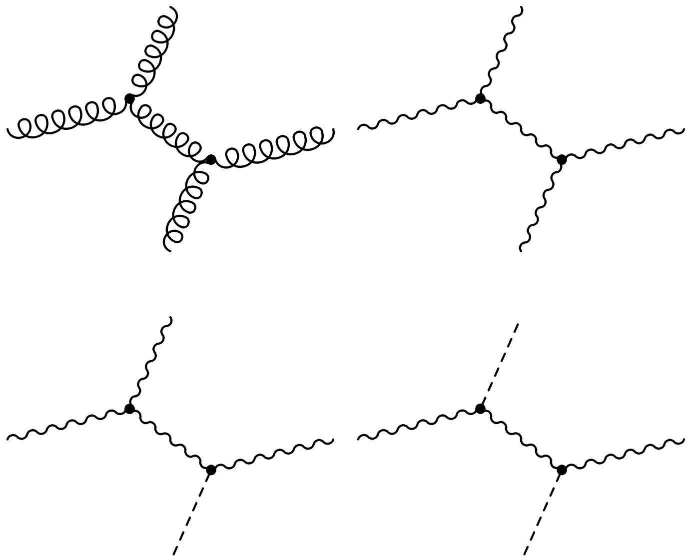


Figure 2: Four boson vertices available within `AMEGIC++`. Note that all boson lines might be on- or off-shell. Scalar and vector bosons are depicted as dotted and wavy lines, respectively, whereas a gluon is marked by a curly line.

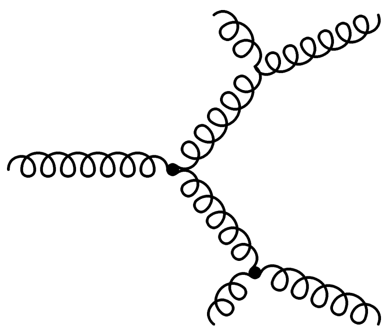


Figure 3: Five gluon vertex available within `AMEGIC++`. Note that all boson lines might be on- or off-shell.

blocks. Instead, we chose to disentangle the gauge structure of such vertices and map them piecewise on corresponding sequences of two three-boson vertices as depicted in the upper left diagram of figure 2.

2.2 Phase space integration

Now we are in the position to discuss the issue of integrating the amplitudes. Due to the high dimensionality of the phase space ($3n-4$ for n outgoing particles) and because of possible non-trivial cuts on the phase space reflecting the experimental situation, we have to abandon the idea of exact analytical integration. Instead we will employ Monte Carlo methods, i.e. we will sample the matrix element, here denoted with $f(\vec{x})$, over the allowed phase space. Its points $\vec{x}$ are then distributed randomly according to the probability

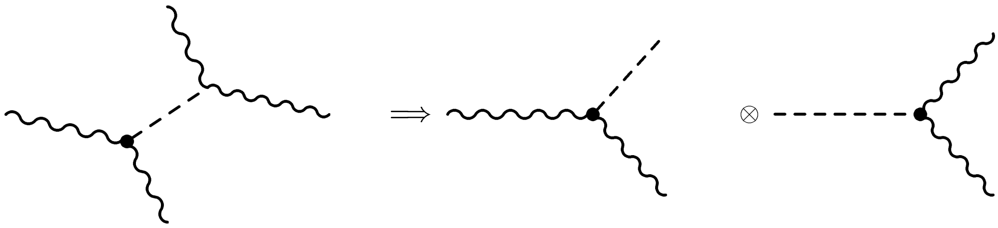


Figure 4: Scalar propagators can be cut easily. Scalar and vector bosons are depicted as dotted and wavy lines, respectively.

distribution $g(\vec{x})$. Defining the weight

$$w(\vec{x}) = \frac{f(\vec{x})}{g(\vec{x})} \tag{2.27}$$

we can transform the true integral into a Monte Carlo estimate $E(f)$ via

$$\int_{\text{true}} d\vec{x}\, f(\vec{x}) = \int_{\text{true}} d\vec{x}\, g(\vec{x}) w(\vec{x}) \Longrightarrow \langle w(\vec{x})\rangle_g = E(f)\,. \tag{2.28}$$

To estimate the error of this estimate $E(f)$, we need the square root of the variance $V(f)$ given by

$$V(g) = \frac{1}{N}\left[\left\langle w^2(\vec{x})\right\rangle_g - \langle w(\vec{x})\rangle_g^2\right] = \frac{1}{N}\left[W(g) - E^2(f)\right], \tag{2.29}$$

where we have identified

$$W(g) = \left\langle w^2(\vec{x})\right\rangle_g. \tag{2.30}$$

The name of the game is now to find a g that efficiently minimizes $V(f)$. This might become quite a tricky business, since usually, the amplitude squared $f(\vec{x})$ is wildly fluctuating due to resonant structures in propagators and — in the Monte Carlo sense — singular behavior at some edges of the allowed phase space. As illustrative examples, consider the phase space for the processes

$$e^+e^- \longrightarrow W^+W^- \longrightarrow 4f\,, \qquad \text{and} \qquad e^+e^- \longrightarrow q\bar{q}g \tag{2.31}$$

which suffer from the resonant W-propagators or from the soft and collinear divergencies related to the gluon emission, respectively. To solve this problem of mapping the phase space efficiently to suitable probability distributions, `AMEGIC++` employs the multi-channel method described for instance in [12]. The cornerstone of this method is the knowledge about the singularity structure of individual processes and the possibility of constructing an efficient mapping as a sum of specific channels. Each of these channels is constructed to cover the full phase space under consideration with special emphasis, i.e. higher probability density, on one of the singular regions. In the following we will briefly review this method and list the ready-to-use standard channels in `AMEGIC++` plus the building blocks for the constructed channels.[3]

[3]The building blocks used in `AMEGIC++` were used already in [14]. Note that similar approaches were discussed in the literature, for example see [15, 16].

2.2.1 Multi-channel method

As indicated above, the basic idea of the multi-channel method is to introduce different mappings of the phase space, i.e. different distributions of the $\vec{x}$. Each of these channels $g_i(\vec{x})$ with $i = 1, \ldots, n$ is non negative over the full phase space and normalized to unity,

$$\int d\vec{x}\, g_i(\vec{x}) = 1\,. \tag{2.32}$$

The various g_i contribute according to so-called a priori weights $\alpha_i \geq 0$ which again are normalized to unity, i.e.

$$\sum_{i=1}^{n} \alpha_i = 1\,. \tag{2.33}$$

Thus, the multi-channel mapping g is

$$g(\vec{x}) = \sum_{i=1}^{n} \alpha_i g_i(\vec{x})\,. \tag{2.34}$$

Since in eq. (2.29) $E(f) = \langle w(\vec{x})\rangle$ is independent of $g(\vec{x})$, we see immediately that we need to minimize $\langle w(\vec{x})^2\rangle_g$ with respect to the α_i in order to minimize our error estimate. It is well known that the overall variance is minimized by distributing it evenly over the channels used. In practice, the essential part of the variance per channel can be easily estimated during the sampling procedure via

$$W_i(\alpha_i) = \int d\vec{x}\, g_i(\vec{x}) w^2(\vec{x}) = \left\langle \frac{g_i(\vec{x})}{g(\vec{x})}\, w^2(\vec{x}) \right\rangle. \tag{2.35}$$

Obviously, α_i should increase, i.e. the channel i should contribute more, if its variance is large, i.e. if W_i is large. In other words, the more wildly fluctuating a channel is, the more Monte-Carlo points using this channel should be produced to smooth out the fluctuations of this particular channel and hence of the overall result. This leads to an iterative improvement of the α_i with help of the W_i.

In AMEGIC++, we therefore iterate an optimization procedure of the α_i with a limited, user-defined number of steps. In each optimization step, i.e. after a predetermined (and again user-defined) number of Monte-Carlo points, the a priori weights are reshuffled according to

$$\alpha_i^{\rm new} \sim \alpha_i^{\rm old} W_i \left(\alpha_i^{\rm old}\right)^{\beta}. \tag{2.36}$$

Again, they are normalized in order to cope with the constraint that they add up to unity. Note that β should obey $0 < \beta < 1$. In AMEGIC++ its default value is 1/2. The optimal set of weights yielding the smallest overall variance during this procedure is stored and used afterwards, i.e. in the full evaluation of the cross section. Note, however, that the results accumulated during the optimization procedure are added to the final estimate.

In the following sections we are going to introduce the channels available within AMEGIC++. They can be roughly divided into two sets, one set of default channels, where the only process dependence manifests itself in the number of momenta produced, the other set consisting of the channels which are constructed by exploring the phase space structure of specific diagrams.

JHEP02(2002)044

2.2.2 Default channels: RAMBO and SARGE

We will start with a brief description of the default channels within AMEGIC++, namely RAMBO and SARGE. For a more detailed discussion of these two integrators we refer to the concise and clear original publications [17, 18].

1. RAMBO:

   Essentially, RAMBO produces any number n of uniformly distributed massless or massive momenta in their center-of-mass frame, i.e. a flat n-particle phase space. Massless momenta are generated in two major steps:

   (a) Energies and solid angles for n "unrestricted" momenta are produced independently, with a flat distribution of the angles in $\cos\theta$, the polar angle, and ϕ, the azimuthal angle. The energies E_i are chosen according to $E_i \exp(-E_i)$ with no constraints whatsoever.

   (b) The n "unrestricted" momenta are rescaled in order to obey the conservation of total four-momentum. This changes both the energies and angles of the unrestricted momenta when they are mapped to the "restricted" momenta.

   For massive momenta another step is now added

   (a) The massless momenta are rescaled once more, in order to bring them onto their mass-shell. In addition, this induces an extra non-trivial weight (i.e. depending on the actual set of vectors), compensating for the rescaling of the momenta.

3. SARGE:

   In contrast, SARGE generates n massless momenta distributed according to the antenna pole structure

$$\frac{1}{(p_1p_2)(p_2p_3)\cdots(p_{n-1}p_n)(p_np_1)}\,. \tag{2.37}$$

   In principle, the momenta are produced again in two major steps:

   (a) Two massless momenta q_1 and q_n are produced in the c.m. frame of the process. Starting from these two momenta the others are produced recursively via "basic antennae" $dA^k_{i,j}$ according to $dA^2_{1,n}dA^3_{2,n}\cdots dA^{n-1}_{n-2,n}$.

   In the expression for the individual $dA^k_{i,j}$, the lower indices i and j denote the two initial momenta and k denotes the generated momentum, respectively. The antennae density is given as

$$dA^k_{i,j} = d^4k\delta(k^2)\Theta(k_0)\frac{1}{\pi}\frac{p_ip_j}{(p_ik)(p_jk)}g\left(\frac{p_ik}{p_ip_j}\right)g\left(\frac{p_jk}{p_ip_j}\right) \tag{2.38}$$

   with

$$g(\xi) = \frac{1}{2\log\xi_{\rm min}}\Theta(\xi-\xi_{\rm min}^{-1})\Theta(\xi-\xi_{\rm min})\,. \tag{2.39}$$

   It should be noted that $\xi_{\rm min}$ can be related to some physical cuts in relative angles of the momenta and transverse momenta.

The algorithm then is to generate k_0 and $\cos\theta$ (the angle between k and one of the two initial momenta) in the c.m. frame of the initial momenta according to dA, distribute the polar angle uniformly and boost the three-momentum system back to the frame where the initial momenta were given. Obviously, this procedure does not satisfy overall four-momentum conservation, thus the next step is mandatory:

(b) The n "unrestricted" momenta q_i are boosted and rescaled in order to obey the conservation of total four-momentum. Again, this changes both the energies and angles of the unrestricted momenta when they are mapped on the "restricted" momenta, but, of course, it leaves the scalar products related to the single antennae invariant.

Let us note in passing that the algorithm outlined above constitutes only the basic version of `SARGE`, improvements outlined by the authors of this algorithm have been implemented into `AMEGIC++`.

2.2.3 Building blocks for additional channels

In addition to the two default channels presented in section 2.2.2, `AMEGIC++` analyses the phase space structure related to the process and constructs automatically one channel corresponding to every individual Feynman diagram. In table 4 we give some insight into the building blocks, propagators and decays, of these channels.

2.3 Higher-order effects

Some higher-order effects have been included in `AMEGIC++` in order to increase the precision of some of the processes to be calculated. The first example is the consideration of Coulomb effects for two charged bosons which have been produced near their mass-shell, section 2.3.1. On the other hand higher-order effects can be included via the running widths of particles, see section 2.3.2.

2.3.1 Coulomb corrections

In principle, Coulomb corrections contribute to all processes with electrically charged particles. They are particularly important, if two particles are created close to each other and move with a low relative speed. Then the long-range interactions alter the corresponding cross section by a possibly sizeable factor $\sim 1/v$ with v the relative velocity.

Within `AMEGIC++`, such corrections are included for the production of W-bosons as in [19] with a factor

$$C^{WW}_{\rm coul} = \frac{\alpha\pi}{2\beta}\cdot\left[1-\frac{2}{\pi}\arctan\frac{|(\beta_M+\Delta)|^2-\beta^2}{2\beta\,{\rm Im}(\beta_M)}\right] \tag{2.40}$$

with

$$\begin{aligned} M_\pm &= p^2_{W^\pm}-M_W^2\,, \qquad \beta_M=\sqrt{\frac{s-4M_W^2+4i\Gamma_W M_W}{s}}\,,\\ \Delta &= \frac{\left|M_+^2-M_-^2\right|}{s}\,, \qquad \beta=\frac{1}{s}\cdot\sqrt{[s-(M_+-M_-)^2]\,[s-(M_++M_-)^2]}\,. \end{aligned} \tag{2.41}$$

1. Propagators :

Propagator	Distribution	Weight
massless	simple pole $1/s^\eta$ with exponent $\eta < 1$	$\dfrac{s_{\max}^\eta - s_{\min}^\eta}{\eta s^\eta}$
massive	Breit–Wigner with mass M, width Γ	$\dfrac{1}{\pi}\dfrac{M\Gamma}{(s-M^2)^2+M^2\Gamma^2}$

2. Decays :

Decay	Distribution $P \to p_i$	Weight
isotropic 2–body	flat, p_i have masses	$\dfrac{2P^2}{\pi\sqrt{\lambda(P^2,p_1^2,p_2^2)}}$
t-channel	$q_1+q_2 \to p_1+p_2$, $p_{1,2}$ massive, $\cos\theta$ between q_1 and p_1 is distributed like $1/(a-\cos\theta)^\eta$ in the rest frame (a depends on mass of the propagator).	
anisotropic 2–body	$\cos\theta$ between P and p_1 in the rest frame of P is distributed like $1/\cos^\eta\theta$, $p_{1,2}$ are massive	$\dfrac{4P^2}{\pi\sqrt{\lambda(P^2,p_1^2,p_2^2)}}\cdot\dfrac{1}{\cos^\eta\theta}\cdot\dfrac{1-\eta}{(\cos^{1-\eta}\theta_{\max}-\cos^{1-\eta}\theta_{\min})}$
isotropic 3–body	flat, only p_2 is massive	$\dfrac{4}{\pi^2\left(\frac{P^4-p_2^4}{2P^2}+p_2^2\log\frac{p_2^2}{P^2}\right)}$

Table 4: Various decays and propagators used within AMEGIC++ for the creation of additional channels. Here, $\lambda(s,s_1,s_2) = (s-s_1-s_2)^2 - 4s_1s_2$.

This factor just multiplies the corresponding amplitudes with a W^+W^--pair for the higher-order correction, i.e. effectively a one plus this factor is multiplied. The amplitudes including the photon are depicted in figure 5, where we have neglected the decay of the W-pair.

2.3.2 Running masses and widths

Possible schemes for the running of quark masses and boson masses and widths, respectively, have been implemented in AMEGIC++. First of all, the LO running quark masses are related to the pole mass $m_{\rm pol} = m(m)$ via the well-known formula:

$$m(\mu) = m(m)\left[\frac{\alpha_S(\mu)}{\alpha_S(m)}\right]^{\gamma_m^{(0)}/2\beta^{(0)}}, \qquad \gamma_m^{(0)} = 6\,C_F\,. \tag{2.42}$$

Here, the running of the strong coupling α_S is used. Note that a leading order running for both, the strong and the electroweak coupling $\alpha_{\rm QED}$ is implemented as well. The second part belongs to the running of the boson widths. Accordingly, a first ansatz uses fixed masses and widths, whereas the other two try some sophisticated schemes for the

JHEP02(2002)044

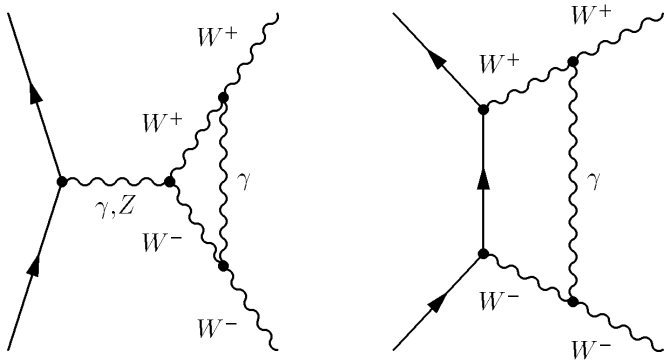


Figure 5: Diagrams related to the Coulomb correction as implemented within AMEGIC++.

Scheme	$m_{\rm fl}(s)$	$\Gamma_{\rm fl}(s)$
0	$m_{\rm pole}$	$\Gamma_{\rm pole}$
1	$\dfrac{m^2_{\rm pole}}{\sqrt{m^2_{\rm pole}-\Gamma^2_{\rm pole}}}$	$\dfrac{m_{\rm pole}\Gamma_{\rm pole}}{\sqrt{m^2_{\rm pole}-\Gamma^2_{\rm pole}}}$
2	$m_{\rm pole}$	$\Gamma_{\rm pole}\cdot\dfrac{\sqrt{s}}{m_{\rm pole}}$

Table 5: Running widths and accordingly modulated masses for the electroweak gauge bosons within the schemes provided by AMEGIC++.

running and appropriate change of these parameters, see table 5. For further details and a comparison between the advantages and disadvantage of these schemes we refer to the literature [20].

3. Algorithms

In this section we describe in some detail the main algorithms implemented in AMEGIC++. We focus on the algorithms employed to generate Feynman diagrams, section 3.1, to translate them into helicity amplitudes, section 3.2, and, finally, on algorithms employed for the construction of suitable channels for the phase space integration, section 3.3.

3.1 Generation of diagrams

Let us start with the generation of Feynman diagrams for a given process. Basically, within AMEGIC++, this proceeds in four steps, namely

1. the creation of empty topologies,
2. setting the endpoints (incoming and outgoing particles),
3. the determination of suitable intermediate propagators, and, finally
4. the elimination of possibly occurring identical diagrams.

In the following, we will discuss these steps in more detail.

3.1.1 Creation of empty topologies

The topologies which are created and then filled by AMEGIC++ during the generation of Feynman amplitudes, consist of binary trees of linked knots, called `Points`. Each of these knots splits into two branches and — correspondingly — two new knots. Therefore, within each point there exist links to a `left` and a `right` point. Consequently, knots which are not splitting any more, point to empty right and left knots. Additional information within a point consists of the flavour of the incoming line and some direction indicated by $b = \pm 1$, depending on whether the line is taken as incoming or outgoing.

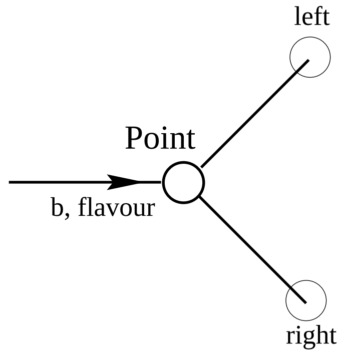


Figure 6.

Let us note that within the Standard Model and its most popular extensions, interactions between particles are described by means of vertices with three and four external legs. In all interesting cases we can understand any four-vertex as two iterated three-vertices with a shrunk internal line. Therefore it is sufficient to use only the binary points for the construction of our topologies.

Consider now processes with $N_{\rm in}$ incoming and $N_{\rm out}$ outgoing particles, respectively. Clearly, starting from a single initial line l_1, we need to construct topologies with $N_{\rm leg} = N_{\rm in} + N_{\rm out} - 1$ additional legs. This can be done recursively, employing all topologies with $n < N_{\rm leg}$. The idea here is to split the initial line l_i into a left and a right branch. The left line l_l then is replaced with all topologies with $l_l \to n$ legs, and the right line l_r with all topologies with $l_r \to N_{\rm leg} - n$ legs. Graphically, this can be understood as depicted in figure 7. Within AMEGIC++ there is a clear distinction of left and right legs due to the internal structure of physical vertices which will be mapped onto the blank topologies in later steps. This is why the sum in figure 7 extends to $N_{\rm leg} - 1$ instead of only half of it.

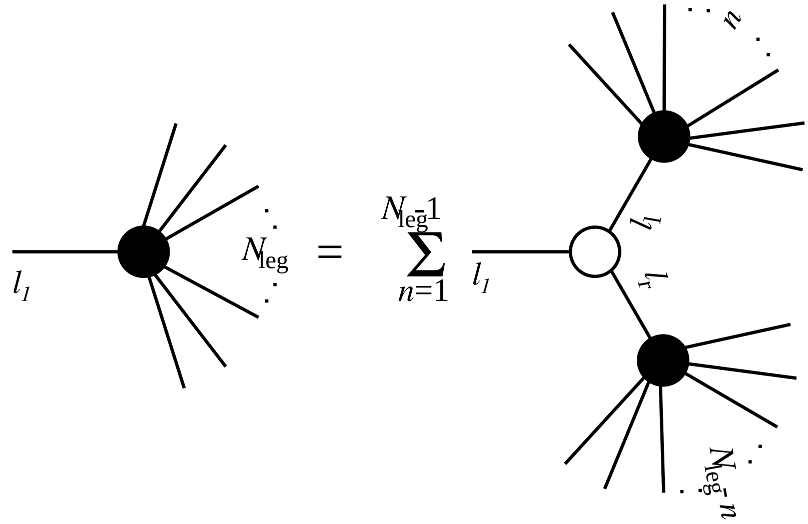


Figure 7: Recursive structure for the construction of topologies.

3.1.2 Setting the endpoints

The next step is to set the endpoints, i.e. to distribute the $n = N_{\text{in}} + N_{\text{out}}$ incoming and outgoing particles over the external legs of the corresponding topologies. A crucial role plays the way how the vertices are set up internally, and how the diagrams will be mapped on the helicity amplitudes. In fact, this allows us to define a number of criteria the endpoints have to meet. First of all, the first incoming particle is set on position 1, i.e. it constitutes the “trunk” of the topology. All other particles are then permuted over positions 2 to n.

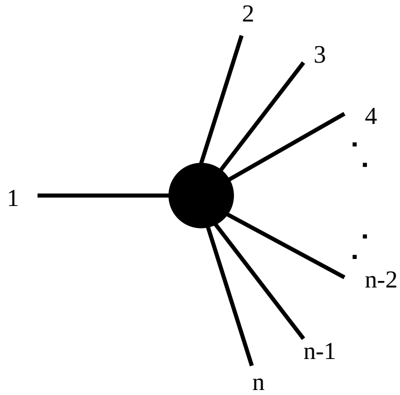


Figure 8.

However, the set of permutations leading to accessible Feynman diagrams is limited by the intrinsic ordering of incoming and outgoing lines in AMEGIC++, see table 6. These limitations[4] are reflected by the following constraints to be applied to the particles on positions 2 and n:

1. If the particle on position 1 is an incoming quark q_{in} (lepton of the kth generation l^k_{in}) the particle on position n has to be either an incoming anti-quark $\bar{q}_{\text{in}}$ ($\bar{l}^k_{\text{in}}$) or an outgoing quark q_{out} (l^k_{out}).

2. If the particle on position 1 is an incoming anti-quark $\bar{q}_{\text{in}}$ (anti-lepton of the kth generation $\bar{l}^k_{\text{in}}$) the particle on position 2 has to be either an incoming quark q_{out} (l^k_{out}) or an outgoing anti-quark $\bar{q}_{\text{in}}$ ($\bar{l}^k_{\text{in}}$).

These two conditions reflect the fact that any incoming fermion on position 1 leads to another fermion (incoming or outgoing) and one boson. Looking up the ordering of particle types in the vertices (as specified in table 6), it is obvious that spinor lines stretch from the incoming particle `in[0]` to the outgoing particle positioned on the right leg `in[2]`,

in[0]	$\rightarrow$	in[1]	in[2]
f_1	$\rightarrow$	b	f_2
$\bar{f}_1$	$\rightarrow$	$\bar{f}_2$	$\bar{b}$
f_2	$\rightarrow$	$\bar{b}$	f_1
$\bar{f}_2$	$\rightarrow$	$\bar{f}_1$	$\bar{b}$
b	$\rightarrow$	$\bar{f}_2$	f_1
$\bar{b}$	$\rightarrow$	f_1	$\bar{f}_2$

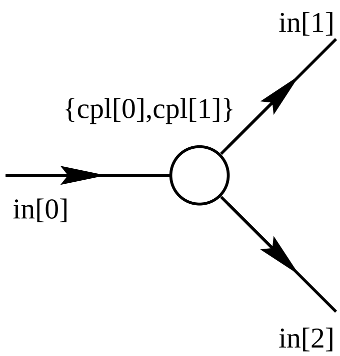


Table 6: Vertices within AMEGIC++. Note (1) the similarity with the point structure encountered before and (2) the specific ordering of fermions and anti-fermions.

[4]Note that at the moment, Standard Model processes only are available within AMEGIC++. However, the inclusion of some popular extensions of the Standard Model into AMEGIC++ is under way. For these, conserved baryon and lepton generation-number will also be assumed at the beginning.

JHEP02(2002)044

whereas anti-spinor lines stretch from `in[0]` to `in[1]`. Thus, in addition with the following algorithms to find intermediate lines (as applied to the boson line attached to the fermion on position 1), the two conditions above guarantee that the existence of a first fermion-fermion-boson vertex is not ruled out *a priori*.

For further refinement, there is some bookkeeping of incoming and outgoing fermions, represented by a variable $q_{\rm sum}$ for the quarks and similarly for each lepton generation. After each position of the external legs is equipped with a particle, $q_{\rm sum}$ is calculated for each permutation according to

$$q_{\rm sum} \longrightarrow \begin{cases} q_{\rm sum}+1\,, & \text{if particle } = \bar{q}_{\rm in} \text{ or } q_{\rm out}\,, \\ q_{\rm sum}-1\,, & \text{if particle } = \bar{q}_{\rm out} \text{ or } q_{\rm in}\,. \end{cases} \tag{3.1}$$

Permutations under consideration are thrown away, if

1. $q_{\rm sum}$ ever exceeds 0 when a quark sits on position 1, and position n is occupied by either an incoming anti-quark or an outgoing quark, or if
2. $q_{\rm sum}$ ever drops below 0, if an anti-quark sits on position 1 and position 2 is occupied by either an incoming quark or an outgoing anti-quark.

The same conditions apply generationwise for the leptons. Whenever a permutation of the external particles meets the requirements outlined above, the endpoints are set accordingly, i.e. they are supplied with the position numbers of the specific permutation under consideration.

Intermediate points receive increasing numbers larger than 99 to distinguish them from endpoints later on. This procedure of giving numbers to the individual points is realized recursively, using the following steps:

1. Start from point 1. Set as its number the number of the first particle.
2. Go to the left point. If it is an endpoint (i.e. with empty left and right links), set the next position of the permutation sequence as its number and proceed to 3. If it is not an endpoint, supply it with an internal number and increment the corresponding counter. Repeat this step.
3. Go back one step and then to the right point. Again, if it is an endpoint, set the corresponding number of the permutation, and repeat step 3. In the complementary case, supply it with an internal number and go to step 2. If there is no right point left, the topology is equipped with all endpoints set.

Note that in this procedure, the position numbers of the permutation are synonymous for the corresponding particle which consists at this stage of a flavour and the flag $b = \pm 1$, where the two signs depend on whether the particle is outgoing or incoming.

To keep track of additional minus signs due to the exchange of two identical fermions, the original numbers of the particles are compared with their permuted numbers. Whenever the ordering of the original and the permuted numbers of two identical fermions does not coincide, the overall sign of the specific combination of endpoints is multiplied by -1.

3.1.3 Finding intermediate lines

Having at hand such a topology with endpoints set and internal numbers for the propagators, the aim is now to find suitable intermediate lines connecting the external points of this specific topology. If any set of intermediate lines is found, a Feynman diagram has been successfully constructed, and if not, the topology can be neglected. Again, the determination of the internal lines proceeds recursively along the following steps:

1. Start from point P with a given flavour. Check whether either the left or the right or both links are not yet equipped with a flavour. If this is the case, test all available vertices for a fit to the known flavours of P and its links. For every fit copy the topology with this new vertex set. If both flavours were already set, go to the next point and start again.
2. If the left link was not yet equipped with a flavour, repeat step 1 with $P \to P_{\rm left}$.
3. If the right link was not yet equipped with a flavour, repeat step 1 with $P \to P_{\rm right}$.

We will now discuss in some detail, how vertices are tested for a fit into the topology. For this it is indispensable to explain briefly, how vertices are defined within `AMEGIC++`. For simplicity we will display only vertices for two fermions and one boson, $f_1 \to f_2 b$ plus their "barred" combinations. Note, however that the same transformations apply also for bosons.

Basically, within `AMEGIC++` there are six different ways to group these three flavours as incoming or outgoing lines, possible double countings due to identical flavours will be eliminated during the initialization of the vertices. In general, during the matching of the vertices onto the topologies, positions `in[0]`, `in[1]`, and `in[2]` will be mapped onto a point, its left, and its right link, respectively. As can be seen in table 6, the fermions are ordered in a specific way, namely the outgoing fermions on position `in[2]` and outgoing anti-fermions on position `in[1]`.

The test of the vertices for a fit into a given topology now proceeds on three levels:

1. The flavours and b-flags of the point under consideration are determined. For simplicity, let us introduce `fl[0]`, `fl[1]`, and `fl[2]` and similarly `b[0]`, `b[1]`, and `b[2]` for the flavours and b-flags ($b = \pm 1$ for incoming and outgoing particles) related to the points P, $P_{\rm left}$, and $P_{\rm right}$, respectively. Then the following adjustments are made (to allow a matching on the vertices):
   - A priori, if points $P_{\rm left}$ or $P_{\rm right}$ are not yet equipped with a flavour, i.e. if they are propagators, their respective b's will be set to 0.
   - `fl[0]` is a boson: if $\texttt{b[1]} = -1$ ($\texttt{b[2]} = -1$), then $\texttt{fl[1]} \to \overline{\texttt{fl[1]}}$ ($\texttt{fl[2]} \to \overline{\texttt{fl[2]}}$). If `fl[1]` or `fl[2]` are not known yet, $\texttt{b[1]} = -1$ or $\texttt{b[2]} = -1$.
   - `fl[0]` is an anti-fermion: if $\texttt{b[0]} \cdot \texttt{b[1]} = 1$, then $\texttt{fl[1]} \to \overline{\texttt{fl[1]}}$. If `fl[1]` is not known yet, then $\texttt{b[1]} = \texttt{b[0]}$.
   - `fl[0]` is a fermion: if $\texttt{b[0]} \cdot \texttt{b[2]} = 1$, then $\texttt{fl[2]} \to \overline{\texttt{fl[2]}}$. If `fl[2]` is not known yet, then $\texttt{b[2]} = \texttt{b[0]}$.

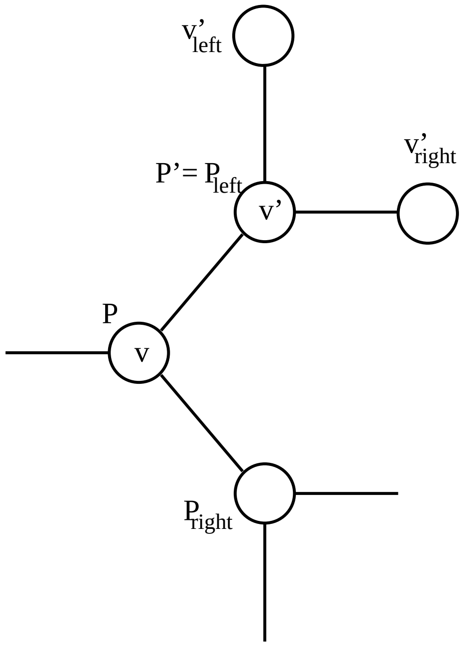


Figure 9: Example topology for the mapping procedure.

2. Now every vertex available is tested against the — modified — flavours of the specific point, i.e. P, P_{left}, and P_{right}. Any vertex v passing this test eventually defines the flavours of internal lines which then will have to fit themselves. The corresponding test is described in the next step.

3. Let us describe this test for flavour `fl[1]` related to the point $P' = P_{\text{left}}$, c.f. figure 9. If both its left and its right link (denoted by P'_{left} and P'_{right}) are endpoints, all available vertices are tested against them after the following transformations:

   - If P' denotes a boson, the flavours of P'_{left} and P'_{right} will be treated as barred and if the corresponding particles are incoming (`b` $= -1$).
   - If P' is an anti-fermion and if `b'[0]`$\cdot$`b'[1]`$= 1$, then the flavour of P'_{left} will be regarded as barred.
   - If P' is a fermion and if `b'[0]`$_{v'}\cdot$ `b'[2]` $= 1$, then the flavour of P'_{right} is barred.

   If now the modified flavours of any vertex v' fit to the flavours of the points P', P'_{left}, and P'_{right} then this vertex makes an acceptable choice. If any such vertex is found, the flavour of P' has passed the test.

   Of course, if either of the two points P'_{left} and P'_{right} is not an endpoint, the test described above is unnecessary.

3.1.4 Eliminating identical diagrams

Unfortunately, the algorithms described in the preceeding sections might produce identical diagrams, mainly because there is no really sophisticated treatment of identical particles and topologies. Within AMEGIC++ therefore, the cure to potential double counting by identical diagrams is explicit comparison of each pair of diagrams. This again proceeds recursively by following the point lists of both diagrams under test with the following algorithm

1. If both points $\mathtt{p}_1$ and $\mathtt{p}_2$ to be compared do not have the same number of offsprings or have different flavour, the recursion terminates.

2. The test of step 1 is repeated with $\mathtt{p}_1 \rightarrow$`left` and $\mathtt{p}_2 \rightarrow$`left`, and possibly after passing with $\mathtt{p}_1 \rightarrow$`right` and $\mathtt{p}_2 \rightarrow$`right`.

3. If the test of step 2 yields a negative result, then there is still the possibility of double counting due to identical particles. To catch this the flavours of $\mathtt{p}_{1,2} \rightarrow$ {`left, right`} are checked crosswise. If this test is passed, i.e. if the flavours of $\mathtt{p}_1 \rightarrow$`right` and $\mathtt{p}_2 \rightarrow$`left` and of $\mathtt{p}_1 \rightarrow$`left` and $\mathtt{p}_2 \rightarrow$`right` mutually agree, step 1 is repeated with both combinations.

4. Finally, if the recursion does not terminate before all points were tested, the diagrams are identical and the second one will be discarded.

3.2 Translation into helicity amplitudes

In this section, we will turn our focus on the translation of the generated Feynman diagrams into corresponding helicity amplitudes and on some items concerning their evaluation once a set of external momenta is given. The following issues will be discussed in some detail:

1. The treatment of the momentum flow along the spinor lines,

2. the piecewise conversion of the diagrams into corresponding Z-functions,

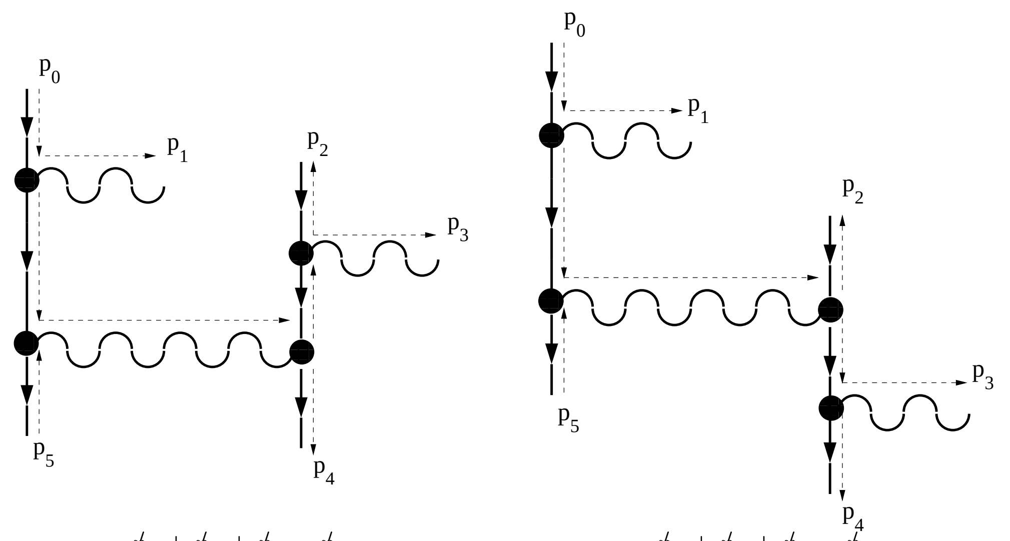
3. and the evaluation of the helicity amplitudes.

3.2.1 Spinor direction and momentum flow

Having generated Feynman diagrams including relative factors of -1 for the exchange of two identical fermions, we are now in the position to construct the corresponding helicity amplitudes. However, there's a caveat here, since another source of relative phases of -1 has not been discussed yet. These additional factors stem from a mismatch of momentum and spinor flow along a line. To illustrate this point, consider the diagrams exhibited in figure 10. Obviously, the sign of the momentum of each fermion line, either external or intermediate, is defined with respect to the spin direction. This results for instance in spinors $\bar{v}(-p_5)$ for the incoming positron in the graphs depicted in figure 10.

Coming back to the construction of Feynman amplitudes described above, the flavours of the point lists are given as in figure 11. For all but the leftmost fermion lines, a general rule can be formulated as follows:

- If a point `p` has an incoming boson as flavour, and if `p` $\rightarrow$ `left` is an intermediate fermion line (with number 100 or larger), then the overall sign of the diagram is multiplied by -1.



$$\bar{v}(-p_5)\gamma_\mu \frac{\not{p}_2+\not{p}_3+\not{p}_4-\not{p}_5}{(p_2+p_3+p_4-p_5)^2}\gamma_\nu u(p_0)\times \qquad \bar{v}(-p_5)\gamma_\mu \frac{\not{p}_2+\not{p}_3+\not{p}_4-\not{p}_5}{(p_2+p_3+p_4-p_5)^2}\gamma_\nu u(p_0)\times$$

$$\times\bar{u}(p_4)\gamma^\mu \frac{-(\not{p}_2+\not{p}_3)}{(p_2+p_3)^2}\gamma_\rho v(-p_2) \qquad \times\bar{u}(p_4)\gamma_\rho \frac{(\not{p}_3+\not{p}_4)}{(p_3+p_4)^2}\gamma^\mu v(-p_2)$$

Figure 10: Example graphs for the process $e^-(p_0)e^+(p_5) \to e^-(p_2)e^+(p_4)\gamma(p_1)\gamma(p_3)$ and the spinorial part of the amplitudes. Clearly, there is a difference in the sign of both diagrams due to the misalignment of momentum and spin-flow in the left diagram. Note that t-channel like topologies emerging from exchanging incoming e^+ and outgoing e^- do not alter the overall signs of the propagators. Instead, they merely lead to a replacement $p_2 \leftrightarrow -p_5$.

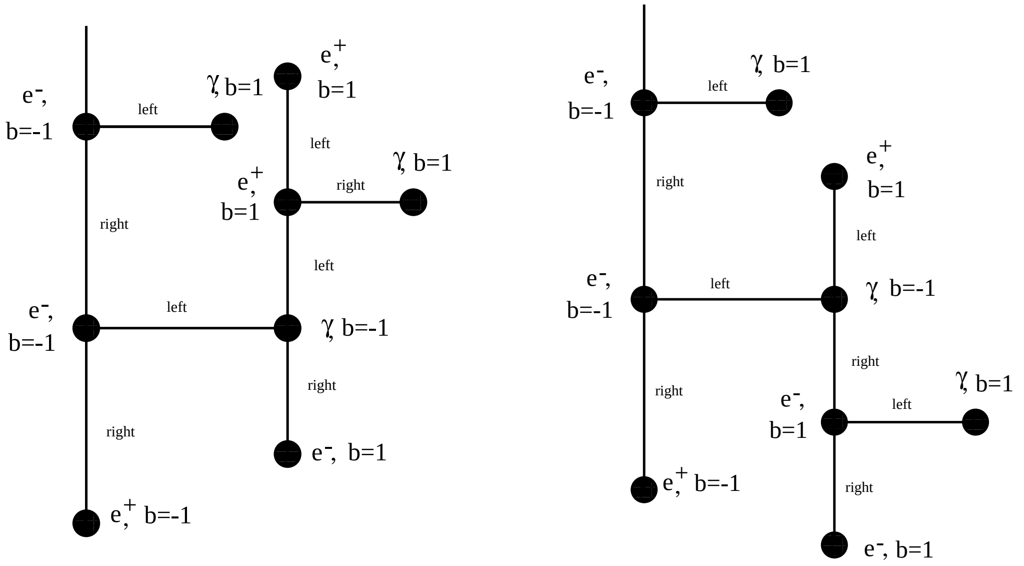


Figure 11: Assignments for the point lists corresponding to the graphs in figure 10. Note that on the level of point lists, the t-channel topologies differ only in their endpoints.

This rule relies only on the fact that for incoming boson lines, there is a strict ordering of outgoing fermion lines with respect to particles/anti-particles. This ordering immediately allows to extract information about the alignment of spin and momentum flow along the lines.

However, the rule above is of somewhat limited use only, since apparently the leftmost fermion line, i.e. the one starting at the point `p[0]` is not dealt with. According to the algorithms employed for the construction of the amplitudes, the first point of this line, `p[0]`, is occupied by an incoming particle.

The leftmost fermion line, i.e. the line starting at the first point, might contribute a change of sign, if it starts with an incoming particle represented by an u-spinor or with an outgoing anti-particle represented by a v-spinor. Then the corresponding endpoint of the spinor line has to be found. This endpoint is then represented by either an $\bar{u}$ or a $\bar{v}$-spinor, i.e. it is either an outgoing particle or an incoming anti-particle.

3.2.2 Finding the appropriate Z-functions

Now we are in the position to present the algorithms leading finally to the helicity amplitudes, or in other words, to the appropriate products of Z-functions. This proceeds in two steps:

1. Projecting the point list onto the larger building blocks, i.e. pieces with a number of intermediate vector and scalar bosons, and
2. flipping the arguments of the emerging Z-functions, such that the particles associated with a "barred" spinor are always followed by particles with "unbarred" spinors. This point applies for the spinors constituting the polarization vectors of spin-1 bosons as well.

We will now elucidate the two steps above. For a better understanding of the projection algorithm mapping the point lists onto helicity amplitudes we recall briefly the structure of the basic building blocks available within `AMEGIC++`. Schematically, they can be represented like in the figure on the right. They all consist of a number of boson propagators — the wiggly lines — between fermion propagators or external spinors, related to fermions or their polarization vectors, represented by the arrowed straight lines. The blob in the middle then represents all possible Lorentz-structures connecting the boson propagators.

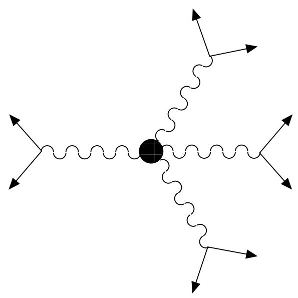
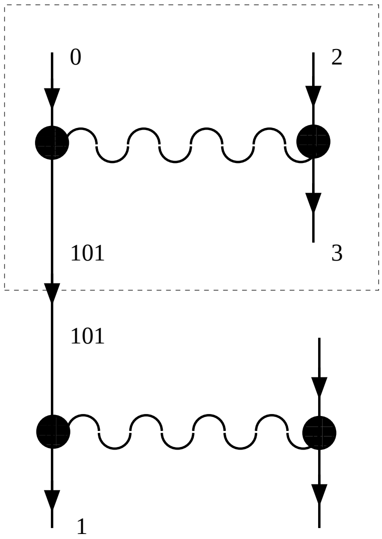


Figure 12.

These building blocks naturally emerge, because intermediate fermion lines can be cut by use of the completeness relation, cf. eq. (2.12), while vector boson lines cannot be cut in such a way. These considerations lead straightforward to the algorithm employed in the mapping procedure sketched below:

1. Start with a point `p`. Check, if the particle related to this point is either an external or an intermediate fermion (with `p` $\rightarrow$ `left` a boson and `p` $\rightarrow$ `right` a fermion), an external or intermediate anti-fermion (with `p` $\rightarrow$ `left` an anti-fermion and `p` $\rightarrow$ `right` a boson) or an external boson. This condition ensures that
   (a) a reasonable translation is possible at all, where the ordering of the connected points for incoming fermions or anti-fermions is just another check, and that



$\Longrightarrow Z_\gamma[101,\, 0;\, 3,\, 2]Z_\gamma[1,\, 101;\, \ldots]$

Figure 13: Example diagram for flipping the ordering of spinors within Z-functions.

(b) intermediate boson lines are not double counted.

If the conditions above are met, the boson is selected and the steps 2 and 3 are executed.

2. Starting from the boson, the links of the subsequent points are followed recursively along possibly occurring further bosons. In every boson line branching into two fermions this recursion terminates. By this procedure, the number of bosons, their type and their topological connections are explored. These characteristics determine which of the building blocks exhibited in figures 1, 2, and 3 are to be used.

3. Having determined the structure of the building block, the task left for the translation of this particular piece is to fill in the corresponding arguments and couplings.

4. The procedure is repeated with the points `p` $\to$ `left` and `p` $\to$ `right`.

Unfortunately, this procedure does not care about any ordering of spinors coming into play either via external particles or their polarizations or by using the completeness relation on the intermediate fermion lines. Let us illustrate this by considering the amplitude depicted in figure 13.

We see immediately that the particle indices occur in ordered pairs of the form $\{\bar{u},\, u\}$. However, within `AMEGIC++`, this ordering of the spinor pieces against the spin flow is mandatory but is not necessarily realized immediately after the construction of the helicity amplitudes. Hence, after being constructed, the Z-functions are searched for external fermions or anti-fermions not obeying the ordering $\bar{u}$, u. If found, the indices within the corresponding pair are switched. If the partner-index denotes a propagator, the index occurs twice due to the completeness relation. Consequently it has to appear exactly once on the first and on the second position of such a pair. This is taken care of by switching the sequence in the other Z-function in which this particular index appears.

3.2.3 Evaluation of the amplitudes

As the last item related to the helicity amplitudes, we discuss briefly, how for a given set of external momenta and helicities a particular amplitude is evaluated. The key point to notice here is that we cut open the fermion lines by use of the completeness relation eq. (2.12). Thus for each intermediate fermion line a sum over both helicities and both particle and anti-particle type spinors ($u\bar{u}$ and $v\bar{v}$, respectively) has to be performed. For massive particles, the masses of the u- and the v-spinors are connected with different signs, entering finally the Z-functions. Therefore, the internal fermion lines are counted first, then for each internal fermion line two summation indices are introduced, labeling the two helicities and the particle and anti-particle components, respectively. For each combination of these indices, the corresponding spinor labels are filled into the Z-functions, and a minus sign internally labels anti-particles and will be recognized at the level of the building blocks to alter the sign of the corresponding mass. Finally, for each combination of internal helicities and particle anti-particle labels all building blocks are multiplied separately to be summed.

3.3 Generation of integration channels

Let us turn now to the generation of additional channels for the multi-channel phase space integration performed by `AMEGIC++`. As already explained in section 2, for the definition of efficient integration channels it is essential to know as much as possible about the structure of the integrand, i.e. possible (in the Monte Carlo sense) singularities in phase space. For `AMEGIC++`, this proves to be the case, because the Feynman diagrams are already at hand. So, basically, the steps performed within `AMEGIC++` for the construction of channels are:

1. For each Feynman diagram the internal lines are identified and it is decided, whether they are in the s- or in the t-channel,
2. then the propagators and decays are translated according to their properties into the building blocks listed in table 4,
3. finally a subset of the individual channels (each corresponding to one specific diagram) is selected.

3.3.1 Properties of the internal lines

The first step in the construction of an integration channel for a particular Feynman diagram is to decide, in which kinematical region the individual propagators are, i.e. whether they are s- or t-channel propagators. This again is done recursively starting from position 1 in the list of linked points constituting the diagram. By iterative steps to the left and right links the other endpoint with $b = -1$, i.e. the other endpoint related to the second incoming particle is found. In these iterative steps, connections to the corresponding previous points are set. It is then straightforward to follow their track back to the starting point, equipping each propagator on the way with a flag indicating that this is a t-channel propagator. More graphically, this identification amounts to a "redrawing the diagram with t-channels running vertically and s-channels running horizontally", see figure 14.

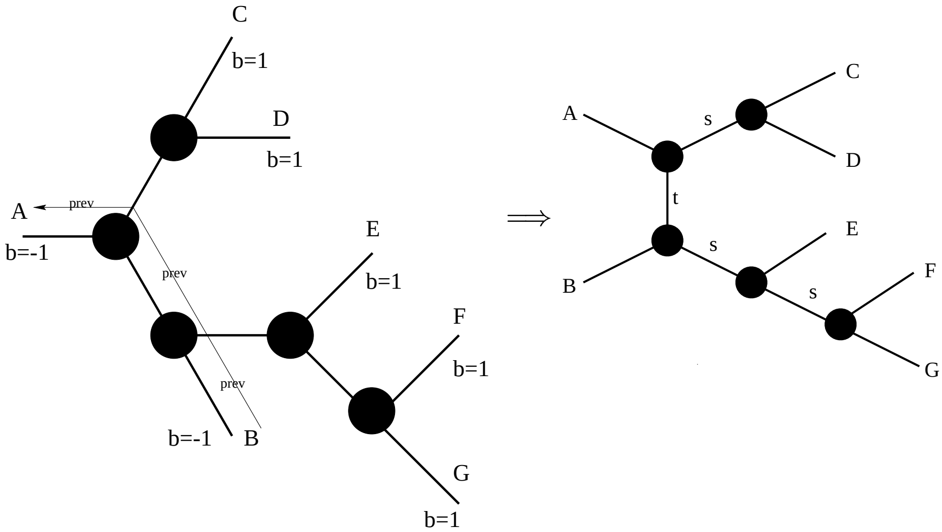


Figure 14: Identifying the propagator types for a process $A + B \to C + D + E + F + G$.

3.3.2 Construction of channels

Now we are in the position to construct appropriate channels. The steps here are the following:

1. Depending on the number of t-channel propagators, the construction of channels in principle starts either with an isotropic two-body decay, with a t-channel propagator or with an isotropic three-body decay (for $n_t = 0$, 1, 2, respectively).

2. The first task when implementing any decay then is to determine the (virtual) masses of the decay products. This is achieved in the following way:

   - First, decay products which are outgoing particles in the process under consideration are put on their mass-shell.
   - Then, decay products which are propagators are equipped with a virtual mass squared s. It satisfies $s_{\rm min} \le s \le s_{\rm max}$ and is distributed either according to a single pole (massless propagators, i.e. propagators with mass $m_p^2 < s_{\rm min}$) or according to a Breit-Wigner distribution (massive propagators, i.e. $m_p^2 > s_{\rm min}$). The limits of the virtual mass squared s are given by the following considerations: $\sqrt{s}$ obviously should be larger than the sum of the masses of all outgoing particles produced in the subsequent decays of the propagators. In turn, $\sqrt{s}$ running in a propagator should be smaller than the energy squared entering the production vertex of the propagator minus the masses of the other propagators which are already chosen, and minus the minimum masses of the propagators with yet undefined masses.

3. After having chosen in the first step the basic form of the channel, i.e. the form of the first decay, the subsequent decays are filled in recursively. By default, for the secondary decays, only isotropic and anisotropic two-body decays are available.

JHEP02(2002)044

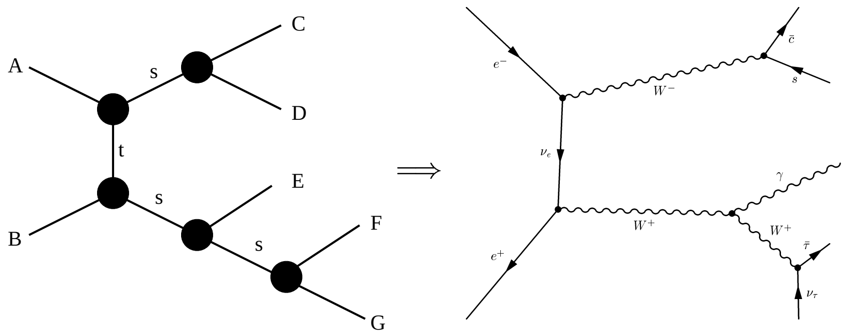


Figure 15: One of the diagrams related to the process $e^+e^- \to s\bar{c}\bar{\tau}\nu_\tau\gamma$.

Again, the momentum transfer squared along the propagators is chosen according to the step above. This procedure of decays and propagators is iterated until only outgoing particles remain.

As an example, let us consider the diagram depicted in figure 15, contributing to the process $e^+e^- \to s\bar{c}\bar{\tau}\nu_\tau\gamma$. Let us assume for better illustration that both propagators CD and EFG are resonating with $M_{CD}\Gamma_{CD} > M_{EFG}\Gamma_{EFG}$ (in fact, in our example process the two products are identical).

The corresponding channel is constructed in the following way:

1. Counting the number of t-channel propagators, we find $n_t = 1$. Therefore the basic form of the channel is determined to be of the simple t-channel form.

2. With $s = (p_A+p_B)^2$ we find the following minimal and maximal momentum transfers:

$$s^{CD}_{\rm min} = (m_C + m_D)^2\,, \qquad s^{CD}_{\rm max} = \left(\sqrt{s} - \sqrt{s^{EFG}_{\rm min}}\right)^2\,,$$

$$s^{EFG}_{\rm min} = (m_E + m_F + m_G)^2\,, \qquad s^{EFG}_{\rm max} = \left(\sqrt{s} - \sqrt{s^{CD}_{\rm min}}\right)^2\,.$$

   Then s^{EFG} is chosen first within its limits as given in the equations above, and only then s^{CD} is determined, with $s^{EFG}_{\rm min}$ replaced by s^{EFG}. These virtual masses for the propagators enter the t-channel decay of step 1.

3. We continue along the CD-propagator. Since both particles C and D are fermions, the decay of $CD \to C + D$ is translated as an isotropic two-body decay.

4. Let us follow now the EFG-line, and consider the radiation of particle E, a photon, off this line. As it is well known, massless vector bosons tend to have an enhancement in the collinear and soft region of emission. Therefore, this decay will be described via an anisotropic two-body decay within the channel.

5. Finally, since particles F and G are fermions, as depicted in our example, their decay is described by an isotropic two-body decay.

The corresponding code generated by `AMEGIC++` for this specific channel can be found in appendix A.

JHEP02(2002)044

3.3.3 Selection of channels

Having generated one channel for each diagram, two things become pretty obvious during integration:

1. Not all diagrams are equally singular, i.e. some of the channels are more effective, whereas others almost do not contribute.

2. Having too many channels, the multi-channel method is not awfully efficient. The various — and often irrelevant — channels are "stealing" a priori weights from each other and from the good, relevant channels.

To alleviate this situation, in `AMEGIC++`, some selection of good channels is performed before the integration starts. Depending on the c.m. energy available for this process, the propagators which yield the singular behavior, are examined for each channel. The maximal number of potentially resonant propagators is determined for each channel, and the most "successful" channels in this category are used in the phase space integration.

4. Implementation

Basically, the determination of cross sections which is the main task of `AMEGIC++`, can be divided into three major stages:

1. The input has to be classified, i.e. the process(es) to be calculated as well as the framework of the model must be determined. At this stage incoming and outgoing particles are specified, particle spectra and couplings are determined, the Feynman rules are established and eventually, using these Feynman rules, particle widths are calculated. Actually, this last step already invokes the other two stages, i.e. generation of Feynman amplitudes and their integration.

2. Now, having established the model framework, the Feynman diagrams related to each single process have to be constructed and translated into helicity amplitudes. Hence, the scattering amplitude is nearly ready for integration. But since some of the helicity combinations yield exactly zero, the calculation can be much alleviated, when these parts are eliminated beforehand. Additionally, the amplitude can be compactified analytically by common factors. These manipulations are performed by translating the expression for the amplitude into a character string and simplifying this character string accordingly. Now, evaluating an amplitude for given external momenta means to interpret a character string. Saving this very string into a `C++` library file and linking this file together with the main program speeds up the calculation considerably. This is the last refining of the amplitudes, and they are ready for integration.

3. Finally, the scattering amplitude has to be integrated by Monte Carlo methods. Hence, the final step consists of the random generation of phase space points, i.e. of sets of four-momenta for the outgoing particles and their summation in order to obtain the cross section. This task could be easily achieved, if the amplitude was

uniform in the whole phase space and would not possess any divergencies in the Monte Carlo sense, i.e. sharp peaks. However, as usual life is much more difficult than one might wish and scattering amplitudes tend to have an abundance of peaks in regions, where particles become either soft, collinear or resonant. The mapping of the uniform and flat distribution of phase space points onto a structure which suits the one of the amplitude, is crucial for the phase space integration.

Following these three major steps, this chapter is structured as follows: in analogy to the first stage, we first describe in section 4.1, how the overall organization of the whole program works out, and how the processes are set up. Then, in section 4.2 we discuss the way, the model under consideration enters. In section 4.3, we then turn our attention to the generation of Feynman diagrams and their translation into helicity amplitudes. The next section, section 4.4, is dedicated to the subsequent treatment of these amplitudes in terms of character strings. Here some of the secrets of the ancient `kabbalistic system` will be disclosed. The final integration over the whole phase space will be discussed in the next section, section 4.5. Last but not least a lot of small helper routines must be provided for accomplishing all tasks. Their explanation as well as the description of the system of parameters and switches will complete this chapter with its last two sections.

4.1 Organization

The central class steering all processes, i.e. the class which is used within the `main` routine, is called unsurprisingly `Amegic`. During the initialization part it specifies the model, i.e. the Feynman rules, starts the calculation of the decay widths and branching ratios of unstable particles, determines all possible topologies up to a certain maximum number of outgoing legs and initializes the individual processes by reading in the incoming and outgoing particles. Therefore it employs all the other major classes of the program, see figure 16.

Specific details of the initialization of the chosen model can be found in section 4.2, whereas the other classes will be described in this section. A brief outline of the classes discussed in this section can be found in table 7.

4.1.1 Organizing the organization — `Amegic`

The class `Amegic` contains the following methods:

1. The `Constructor` of this class runs the whole initialization procedure of the program along the following steps:

   - The model file which has been chosen from the input parameters, is specified and initialized. During this stage the spectrum of the model is determined.
   - The vertices which are in fact the Feynman rules deduced by the model are initialized.
   - The processes to be calculated are read in from the `Process.dat` data file with the help of `Read_Processes()`. The incoming and outgoing particles for the different processes are fixed.

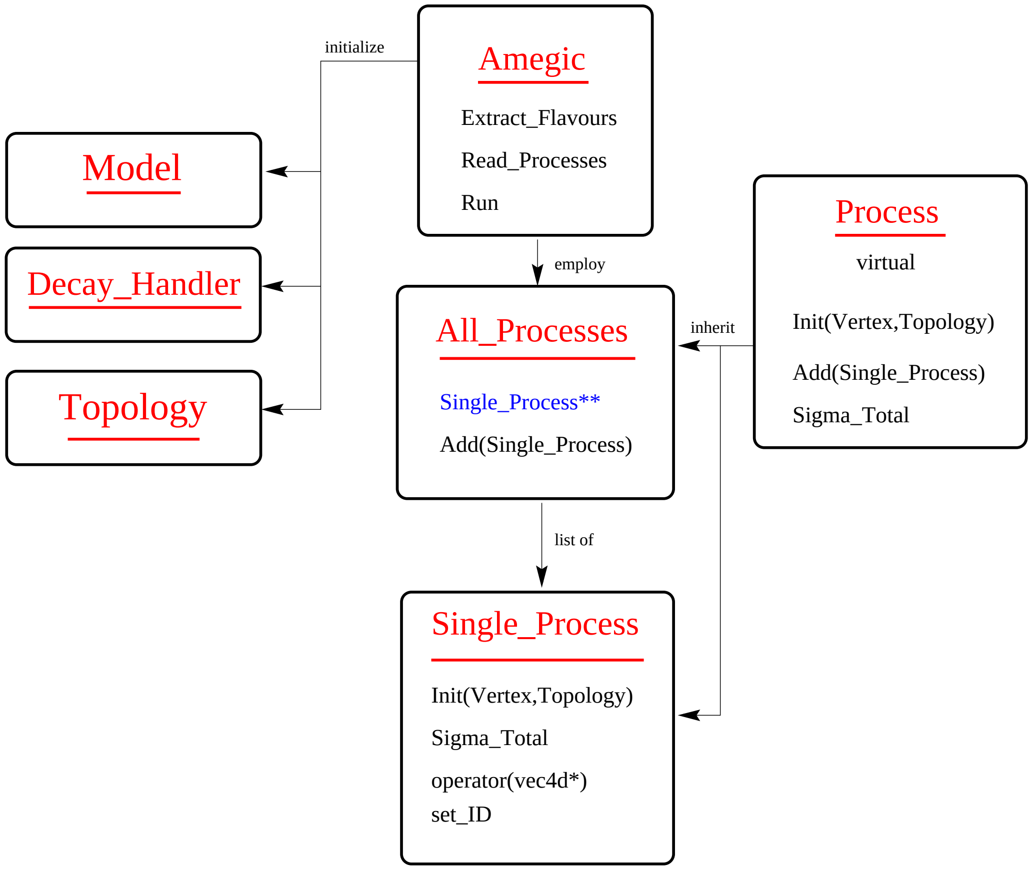


Figure 16: Main structure of the program.

- Having at hand the maximum number of external legs the raw topologies can be determined by constructing a `Topology` object.
- For every particle which is marked as unstable in the `Particle.dat` data file, the total width and the branching ratios have to be calculated using the `Decay_Handler`. With `Decay_Handler::Find_Unstable()` all unstable particles are detected and listed in a `DecayTable`. Now, `Decay_Handler::Calculate()` evaluates the width and the branching fractions of the particles.
- Last but not least the individual processes will be initialized via `Process::Init()`. Consequently the raw topologies are combined with the vertices yielding the Feynman diagrams which are then translated into helicity amplitudes.

2. `Read_Processes()` reads in the list of processes from `Process.dat`, extracts the flavours with the help of `Extract_Flavours()`, and groups the processes into `All_Processes`. If only one process is to be considered, a `Single_Process` will be created instead.

3. In `Extract_Flavours()` the raw text given in the `Process.dat` data file is translated into the numbers and flavours of the incoming and outgoing particles.

Class/Struct	Purpose
Amegic	Main class, reads in the process parameters.
Decay_Handler	Handles all decays of the unstable particles, calculates widths and branching ratios.
DecayTable	The table of decay chains.
Process	Mother class of all processes.
All_Processes	Maintains a number of processes.
Single_Process	Maintains the initialization and calculation of one single process.
Amplitude_Base	Container for all integrable amplitudes.
Helicity	Determines the helicity combinations yielding a non-zero result. Explores and uses symmetries, i.e. identical results for different helicity combinations.
struct signlist	List of helicity signs, contains on-switches.
Polarisation	Provides the polarisation vectors of external bosons.
Point	A point structure for building binary trees.
Topology	Calculates all possible topologies.
Single_Topology	Contains a single topology.

Table 7: A short description of the major classes handling the organization of `AMEGIC++`.

4. `Run()` calculates the total cross section of all involved processes via `Process::Sigma_Total()`.

4.1.2 Decays — `Decay_Handler`

The primary aim of the `Decay_Handler` is to generate a `DecayTable` containing all information concerning the decays of unstable particles, i.e. total widths, branching ratios, the flavour of the decay products etc. The `DecayTable` is constructed as a two-fold list in the following way. Every entry into the `DecayTable` itself is a `DecayTable`, having pointers into two directions (see figure **??**):

1. `Next`, pointing to the next decaying particle or the next decay channel, and

2. `More`, pointing to a list of decay channels for the same decaying particle.

Furthermore, each `DecayTable` has a varying (limited by the user) number of flavours, the flavour of the decaying particle plus the flavours of its decay products. The user defines the maximal number of particles created in a decay. If, in the process of setting up the `DecayTable`, a particle decays into a final state consisting of one or more unstable particles, these can be then replaced by their own decay products.

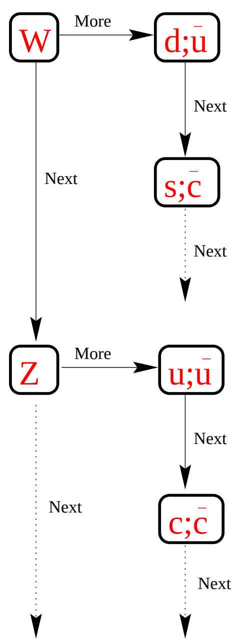


Figure 17.

Now we are in the position to understand, how the `Decay_Handler` fills this table via the following methods:

1. `Find_Unstable()` produces the list of all unstable particles, where the decay width has to be calculated. A simple loop over all particles probes their on-switch which is read in from the `Particle.dat` data file, i.e. whether the user wants to include them or not. Discarded particles are neglected completely, so they do not show up, neither as internal nor as external lines. Having at hand all decaying particles the products will be determined with the routines `Find_Decay_Products_2()` for the $1 \to 2$ decays and `Find_Decay_Products_3()` for any further decays.

2. `Find_Decay_Products_2()` finds every $1 \to 2$ decay for the unstable particles. Naively, one could take every possible pair of particles as decay products. But applying a number of constraints, only decays with non-zero width are taken into account. Thus every pair of decay products has to obey the following restrictions:

   - The incoming flavour has to interact at all, so it has to show up in the list of vertices as an incoming or outgoing particle, respectively. This test is performed with `Check_In_Vertex()`.
   - Due to energy-momentum conservation none of the decay products can be the same as the incoming particle.
   - The masses of the decay products must be separately smaller than the mass of the incoming particle. The sum of both masses can be larger, if one or both of the decay products is unstable and can be understood as an internal line. Later on, the decays into such virtual particles is taken into account via three or more body decays.
   - All quantum numbers, i.e. charge and spin, have to be conserved. Hence, a decay of a fermion into bosons only is disallowed.

   If it fulfills all these constraints the decay channel is added to the `DecayTable` with the help of `Add`, i.e. another item is added to the `More` list of the corresponding decaying particle.

3. In `Add()` the last check for a particle and its decay products is performed. With the help of `Check_Vertex()` the three flavours are compared with the flavours of all existing vertices in the actual model. Only if this check is passed, a new `DecayTable` is added to the list.

4. `Find_Decay_Products_3()` finds any further decay beyond two-body decays. Therefore a loop over all already determined decays is performed. In the case a virtual particle is found, i.e. the sum of the masses of all decay products is larger than the mass of the incoming particle, a further decay of unstable decay products is realized with `Next_Decay()`.

5. `Next_Decay()` is used for the determination of further decays. Now, every decay product is analysed once more. If it is an unstable particle itself, one can of course find it in the list of decaying particles. A further loop over all possible decays of this particle and its replacement by these decay products completes the creation of a new decay mode which then will be attached to the `DecayTable`.

6. `Check_In_Vertex()` checks, if a certain flavour is included in the list of vertices.

7. `Check_Vertex()` checks, if a vertex with the given flavour combination exists.

8. `Calculate()` calculates the width of the different decay modes and sets the branching ratios. It uses the list of decays which has already been constructed in `Find_Unstable()`, to determine the width of the different decay modes using `Rec_Calc()`. Having at hand all decay widths it is straightforward to calculate the different branching ratios with `Branching_Ratios()`.

9. `Rec_Calc()` evaluates the widths of the different decay modes recursively in order to take into account the further decay of virtual particles. For the calculational part, the routines of `Process` are utilized.

10. `Branching_Ratios()` performes a loop over all decaying particles and determines the branching ratios with the help of `Rec_BR()`.

11. `Rec_BR()` calculates the branching ratios recursively, since all further decays have to be taken into account.

12. `Print()` prints the table of decays including the decay widths and branching ratios.

4.1.3 Helicities and polarisations

The structure `signlist` contains one helicity combination, an `On`-switch and a multiplicity. The latter one can be used to reduce the number of helicity combinations by dropping equivalent ones, i.e. those yielding the same numerical result. The organization is done by the class `Helicity` via the following methods:

1. The `Constructor` determines all helicity combinations using a loop over loops technique as described in appendix C. Having at hand all possible helicity combinations with two helicities per outgoing particle, the second helicity for scalar particles (which have no helicity at all) and massive vector bosons (where the sum over the polarizations is carried out differently to the massless vector bosons) is switched off.

2. `Max_Hel()` returns the maximum number of helicities.

3. `operator[]()` returns a certain helicity combination.

4. `switch_off()` switches a certain helicity combination off.

5. `On()` returns the status of a certain helicity combination.

6. `Inc_Mult()` increases the multiplicity of a combination by one.

7. `Multiplicity()` returns the multiplicity of a certain helicity combination.

The class `Polarisation` maintains all methods needed in connection with polarisations for massless and massive vector bosons:

1. `Spin_Average()` The normalization constant for averaging the spin, the color and the polarisations of all incoming particles is determined at this stage. Accordingly, every fermion yields a factor of 1/2, every quark an extra factor of 1/3, every gluon a factor of 1/8 and the massless and massive vector bosons gain a factor of 1/2 and 1/3, respectively.

2. `Massless_Vectors()` determines whether there is any massless vector boson in the in- or outstate. If this is the case, an extra four-momentum vector for the construction of the polarisation vector is attached to the list of vectors.

3. `Massive_Vectors()` counts the number of massive vector bosons in the in- and outstate. For every hit two new four-momenta are attached, see eq. (2.18). The change of the normalization is performed according to eq. (2.20).

4. In `Attach()` a connection between the massive vector boson and its two polarisation vectors is drawn, ensuring that the latter ones add up to the vector boson momentum.

5. `Reset_Gauge_Vectors()` changes the extra gauge vector, when massless vector bosons are present. Usually, this routine is used during the gauge test.

6. `Set_Gauge_Vectors()` sets the gauge vector of the massless vector boson and calculates the two polarization vectors for each massive vector boson. The latter ones are determined according to eq. (2.18) along the following algorithm:
   The four-momentum of the massive vector boson is boosted into its rest frame and two massless four-momenta are generated as uniformly distributed massless pseudo-decay products of the vector boson. Afterwards they are boosted back into the lab frame.

7. `Massless_Norm()` calculates the normalization due to massless vector bosons, see eq. (2.16).

8. `Massive_Norm()` yields the normalization due to massive vector bosons, see eq. (2.20).

9. `Replace_Numbers()`: during the translation of the Feynman diagrams into helicity amplitudes all polarisation vectors obtain dummy numbers. The correct connection to the appropriate polarisation vector is performed in this routine utilizing `Single_Amplitude::MPolconvert()`. As described above, massive vector bosons are treated as propagators decaying into their — massless — polarisation vectors, the respective changes are carried out with the help of `Single_Amplitude::Prop_Replace()`.

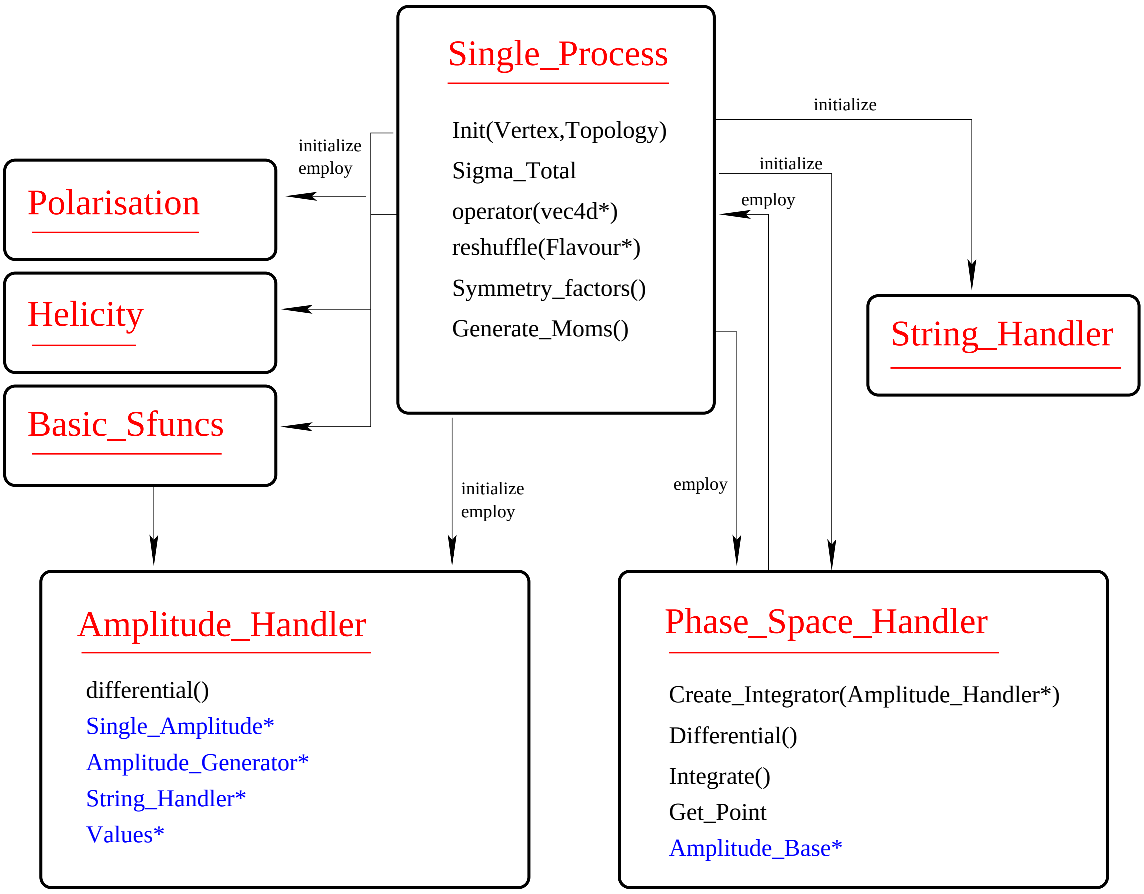


Figure 18: One single Process containing an amplitude and a phase space evaluation.

4.1.4 Handling of processes

The two main classes concerning the overall handling of processes are `All_Processes` and `Single_Process`. Both classes provide the amplitudes squared, either for a list of processes or for a single process, respectively. In order to integrate both with the `Phase_Space_Handler` they have to be derived from the class `Amplitude_Base` which is the base of an (phase space) integrable class. Therefore, its purely virtual operator `operator()()` must be overwritten by the appropriate inherited class. On the other hand, the two process classes have to provide a similar interface which simplifies the usage from outside. Accordingly, both classes are derived from a mother class called `Process`, a virtual class which contains a minimal set of routines, i.e. `Init()`, for the initialization, `Sigma_Total()` for the calculation of the total cross section and `Add()` for adding a new process to `All_Processes`. Since all these methods have to be overwritten by the derived class, a detailed description can be found there.

The container for a list of processes is `All_Processes`:

1. Within the `Constructor()` the maximum number of processes is set and a blank list of `Single_Processes` is generated.

2. `Init()` performs a loop over all processes and initializes them via

Single_Process::Init().

3. Sigma_Total() evaluates the individual cross sections for all processes and adds them up. At this stage a parallelization of the calculation for the different processes would enhance the performance. This is part of a future project.

4. The operator()() is inherited from Amplitude_Base and is used for integrating the sum over all cross sections. Accordingly a loop and a sum over all Single_Process::operator()()s is done.

5. Add() appends another process to the list.

The organization, i.e. the initialization as well as the calculation of the amplitude squared for single process lies in the hands of the class Single_Process:

1. The Constructor() of the class covers a large part of the initialization procedure which is accomplished along:

   - The sequence of the flavours for the incoming and outgoing particles is reshuffled using the routine reshuffle().
   - A string which identifies the given process unambiguously, is generated via set_ID().
   - A directory is created named with this ID. It will contain all the output from this process.
   - The overall normalization is calculated using Symmetry_factors() and Polarization::Spin_Average() taking into account the symmetry factor due to permutations of outgoing particles and the average over the incoming quantum numbers, respectively.
   - The number of polarization vectors is determined, exploring the massless and massive vector bosons with Polarization::Massless_Vectors() and Polarization::Massive_Vectors(), respectively. Now, the number of four-momenta is simply the sum of the number of incoming and outgoing particles plus the number of polarization vectors needed.
   - With Polarization::Attach() the polarization vectors of the massive vector bosons are connected with the appropriate vector boson.
   - The normalization due to the special treatment of massive vector bosons is calculated via Polarization::Massive_Norm().
   - Now, all helicity combinations are determined by constructing the Helicity object.
   - The basic functions for the determination of the building blocks, i.e. the S-functions (see eq. (2.23)), will be initialized by constructing a Basic_Sfuncs object.
   - Last but not least the String_Handler for the generation and translation of the helicity amplitudes into character strings is constructed.

2. Now, the method `Init()` has only a small task left, since the bulk of initializations has been already carried out. This task is the construction of the `Amplitude_Handler` object and the initialization of the S-functions and the `String_Handler` via the appropriate routines `Basic_Sfuncs::Initialize()` and `String_Handler::Init()`, respectively.

3. In `reshuffle()` the user-given sequence of the input flavours is changed to an intrinsic standard. The internal order in `AMEGIC++` is fermions before bosons. Then, particles are first ordered according to the absolute value of their kf-code and particles are always listed before anti-particles. Massless vector bosons are listed at the end of the line which is closely related to the treatment of polarization vectors within the program.

4. `set_ID()` constructs the uniquely defined process name. The simple rule is that at first the number of incoming and outgoing particles is translated to the character string. At next the flavours of these particles follow. An example can be found in the Test Run Output, see appendix D.

5. The `Symmetry_factors()` for outgoing particles are constructed from the multiplicity n of a flavour, where every multiple occurrence gains a factor of $1/n!$.

6. `Tests()` contains two different tests of the derived amplitude. First of all, a gauge test can be performed, if massless vector bosons are present. In this case, the amplitude is calculated with two different choices of the corresponding gauge vectors and the results will be compared. Of course, if the program is doing everything correct, this test should be passed, i.e. both results should coincide for every single combination of external, physical momenta. Taken by itself, this test provides a powerful tool for checking amplitudes, vertices etc. Therefore it might be useful sometimes to add a photon or a gluon to a process to have this kind of test at hand. The next test concerns the translation of the helicity amplitudes into character strings. During this translation numerically small parts will be neglected and the emerging string will be reorganized and simplified. Therefore, a test for the correctness of the string is urgently necessary. Again, the amplitudes calculated with and without the string method are to be compared.

7. `Sigma_Total()` calculates the total cross section for the given process utilizing the class `Phase_Space_Handler`. After constructing this object the method `Phase_Space_Handler::Create_Integrator()` builds all the possible integration channels. With `Phase_Space_Handler::Integrate()` the calculation of the total cross section is achieved.

8. The `operator()()` yields the amplitude squared for a given configuration of four-momenta for the incoming and outgoing particles. The first step in this calculation consists of setting the gauge vector. Then, a loop over all possible helicity combinations is performed and the different values for the individual amplitudes are

summed up. Multiplying the result with the normalization for the gauge vector derived from `Polarization::Massless_Norm()` and the overall normalization gives the amplitude squared.

4.1.5 Construction of topologies

The class `Point` contains all information about a specific vertex, i.e. pointers to the `left` and to the `right` next `Point`, the flavour, the particle or propagator number etc. In this way a tree structure can be generated by means of linked `Points`. Accordingly, all topologies with the same number of legs are contained in a `Single_Topology`, other variables are the number of legs, the depth as well as the list of `Point` lists. The class `Topology` creates and handles a list of `Single_Topologys` (which differ by the number of legs) via:

1. The `Constructor()` constructs all topologies with the method `Build_All()`.

2. `Build_All()` generates all topologies up to a maximum number of legs. It starts by initializing the first simple topology with one external leg only. Since every topology contains all other topologies with a smaller number of external legs, it is created recursively out of those by calling `Build_Single()`.

3. `Build_Single()` builds all topologies for a certain number of external legs, for details concerning the algorithm see section 3.1.

4. `Get()` returns a list of topologies with a certain number of legs.

5. `Copy()` copies a given point list into another.

6. `Print()` prints a topology.

4.2 The model

Cross sections are calculated within the framework of a specific model, i.e. they depend on a set of assumptions and parameters like the particle spectrum and the properties (quantum numbers) of the particles as well as their interactions defined via vertices and the corresponding coupling constants. Strictly speaking, this framework is defined by the underlying lagrangian, from which the parameters can be either read off or calculated. However, for the sake of implementing such a model in terms of computer algorithms, it is sensible to divide the information defining the model into several pieces, reflected by the class structure. In general, in `AMEGIC++` particle properties are handled by the class `Flavour`, interactions are governed by the classes `Model` and `Vertex`, and the appropriate (possibly running) coupling constants and the particle masses are generated by `Coupling` and `Spectrum` classes, respectively.

A brief outline of all the classes mentioned above, can be found in table 8. The different methods of the classes are described in detail in the subsequent sections, covering the complexes Flavour, Vertices, Model, Spectrum and Couplings.

Class/Struct	Purpose
kf	Connects the kf-code of every particle with a name.
kf_to_int	Translates the kf-code into an integer value.
part_info	Contains all the information about a particle which is read in from the `Particle.dat` data file.
Flavour	All particle properties are handled from here.
fl_iter	An iterator over all possible flavours is often useful.
Single_Vertex	Contains all information about a Feynman rule.
Vertex	Maintains the initialization of the vertices, i.e. the Feynman rules of the chosen model.
Model	Is the virtual mother class of all models.
Model_QCD	Contains the QCD Feynman rules.
Model_EE_QCD	Contains the QCD Feynman rules plus a coupling of the QCD particles to electron and positron.
Model_EW	Contains all electroweak Feynman rules.
Model_SM	The complete Standard Model can be found here.
Spectrum_EW	The electroweak spectrum, i.e. all masses of the leptons and gauge bosons are maintained here.
Spectrum_QCD	The QCD spectrum, i.e. the masses of the quarks is read in within this class.
Couplings_EW	The electroweak coupling constants as well as their running is performed within this class.
Couplings_QCD	The running of the strong coupling constant is calculated.

Table 8: Principal classes to set up and organize the physical framework.

4.2.1 The flavour

`kf` contains the kf-code of every particle according to the PDG [21], whereas the small class `kf_to_int` translates this kf-code into an integer value and back. All necessary particle information is stored in `part_info`, namely the kf-code, the mass and width, the charge and the isoweak charge, a flag for strong interacting particles, the spin, a flag for the majorana character of a particle, an on-switch, a flag for stable particles and the name of the particle. The class `Flavour` handles all properties a particle could have. Therefore, it mainly provides routines for obtaining information about a particle type:

1. `texname()` gives the name of a particle in a LaTeX format. This is used for instance by `Vertex::Tex_Output()`.

2. `kfcode()` gives the kf-code of a particle.

3. The `operator int()` returns the integer value of a kf-code. A minus sign covers the possible anti-particle nature.

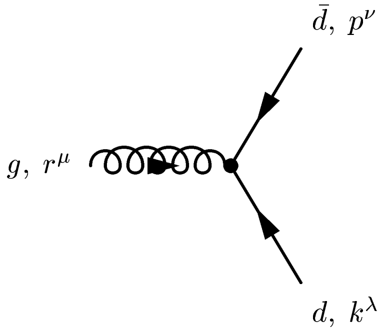
4. `bar()` returns the anti-particle of a particle.

5. `charge()` returns the electromagnetic charge in units of the proton charge.

6. `icharge()` returns as an integer three times the electromagnetic charge.

7. `isoweak()` returns the value of T_3, the third component of the weak isospin.
8. `strong()` yields 1, if the particle interacts via the strong force.
9. `spin()` returns the spin of a particle.
10. `ispin()` returns as integer twice the spin of a particle.
11. `ison()` returns 1, if a particle should take part in the generation of the Feynman rules.
12. `isstable()` is the switch which returns 1 in case the particle width should be calculated by the program, otherwise the width stored within the `Particle.dat` data file is used.
13. `set_on()` switches a particle on.
14. `mass()` returns the mass of a particle.
15. `set_mass()` sets the mass of a particle.
16. `width()` returns the width of a particle.
17. `set_width()` sets the width of a particle.
18. `name()` returns the character name of a particle.
19. `isfermion()` returns 1, if the particle is a fermion.
20. `isboson()` returns 1, if the particle is a boson.
21. `isscalar()` returns 1, if the particle is a scalar, i.e. if it has spin zero.
22. `isvector()` returns 1, if the particle is a vector, i.e. if it has spin one.
23. `isquark()` returns 1, if the particle is a (anti-) quark.
24. `isgluon()` returns 1, if the particle is a gluon.
25. `islepton()` returns 1, if the particle is a (anti-) lepton.
26. `isuptype()` returns 1, if the particle is up-type. Note that this extends to leptons and neutrinos as well.
27. `isdowntype()` returns 1, if the particle is down-type. Note that this extends to leptons and neutrinos as well.
28. `isanti()` returns 1, if it is an anti-particle.
29. `hepevt()` returns the HEPEVT kf-code.
30. `from_hepevt()` sets the flavour according to the HEPEVT kf-code.

The class `fl_iter` provides an iterator over all particles in the particle data table. Two methods, i.e. `First` and `Next` return the first and the appropriate next particle, respectively.



$$P_R i(-g_3) + P_L i(-g_3)$$

Figure 19: A sample vertex in LaTeX output form. Note that P_R and P_L are the right and left-handed projectors including a γ_μ.

4.2.2 The vertices

A `Single_Vertex` contains all information about an $1 \to 2$ Feynman rule, i.e. the three flavours, the left- and right-handed coupling constants and a possible representation by a character string. The class `Vertex` handles the generation of Feynman rules utilizing some of the methods from the class `Model`.

1. The `Constructor()` builds the list of vertices (`Single_Vertex`) and produces a LaTeX output file. First of all, the list is filled via the Model routines `c_FFV`, `c_FFS`, `c_VVV`,`c_SSV`, `c_VVS` and `c_SSS`, see table 9. Having at hand all the different vertices, the LaTeX output is generated with `Tex_Output()`. Since during the matching of the vertices onto the raw topologies not only the standard form of a vertex is used, but also its rotations (obtained as a combination of permutating the flavours and taking their anti-flavours), every vertex is transformed into the six possible combinations with an appropriate change of the couplings, see table 6. A test with `Check_Equal()` prevents any double counting of already generated vertices.

2. `Check_Equal()` compares any combination of three flavours generated in the process of initialization with all already generated vertices and prevents any double counting.

3. `Print()` prints the list of vertices.

4. `operator[]()` returns a `Single_Vertex` from its number.

5. `Max_Number()` returns the maximal number of vertices.

6. `Tex_Output()` generates a LaTeX output of all Feynman rules. The character strings generated for each vertex in the model classes will be combined with a FeynMF [22] picture of a vertex, for example see figure 19.

7. `AddVertex()` adds a `Single_Vertex` to the list of vertices.

8. `FindVertex()` searches for a given `Single_Vertex` in the list of vertices and returns its number.

JHEP02(2002)044

Method	Incoming	Left Outgoing	Right Outgoing
c_FFV()	Fermion	Vector boson	Fermion
c_FFS()	Fermion	Scalar boson	Fermion
c_VVV()	Vector boson	Vector boson	Vector boson
c_SSV()	Scalar boson	Vector boson	Scalar boson
c_VVS()	Vector boson	Scalar boson	Vector boson
c_SSS()	Scalar boson	Scalar boson	Scalar boson

Table 9: Connection of methods for the initialization of vertices with the external particles.

4.2.3 The model

The `Model` class represents the basis for all other model classes which are simply derivatives. Consequently, most methods are purely virtual. However, they play a similar role in every model and therefore, they will be described only once. The first part is dedicated to the initialization of the model and the vertices:

1. `Init_Vertex()` generates the list of vertices by constructing the `Vertex` object.
2. `Get_Vertex()` returns the list of vertices.
3. Usually `Init()` generates the masses and couplings of the given model with the help of an appropriate `Spectrum` and `Coupling` class, respectively.

All Feynman rules can be classified according to their Lorentz structure which depends on the incoming and outgoing flavours. This classification scheme of the vertices in `AMEGIC++` is depicted in table 9, the methods listed there are then filled in a model dependent way.

Since the Model class contains not only all the Feynman rules, but also the different coupling constants some helper methods permit to gather this information:

1. `SinTW()` returns the $\sin\theta_W$.
2. `CosTW()` returns the $\cos\theta_W$.
3. `Aqed()` returns $\alpha_{\rm QED}$ at the CM energy scale.
4. `Aqed(double)` returns $\alpha_{\rm QED}$ at the given scale.
5. `Aqcd()` returns $\alpha_{\rm QCD}$ at the CM energy scale.
6. `Aqcd(double)` returns $\alpha_{\rm QCD}$ at the given scale.

The different model classes which are derived from the main class `Model`, can be characterized via the Feynman rules (i.e. vertices) they initialize:

1. `Model_QCD` contains all pure QCD Feynman rules, i.e. the interaction of quarks via gluons and the gluon self-interaction in the axial gauge.
2. `Model_EE_QCD` includes all vertices from `Model_QCD` plus the interaction of quarks with electrons and positrons via photons and Z bosons.

3. `Model_EW` includes all electroweak vertices, i.e. the interaction of all Standard Model particles via photons, Z, W and Higgs bosons.

4. `Model_SM` combines `Model_QCD` and `Model_EW` to yield all interactions of the Standard Model.

4.2.4 The spectrum

The two different spectra of the Standard Model, i.e. the strong and the electroweak sector, are represented by the classes `Spectrum_EW` and `Spectrum_QCD`. Both have only the method `Fill_Masses()` in order to set or calculate the masses of the different particles, when called. However, within the Standard Model, all fermion masses are fundamental parameters, therefore, they are only read in from the `Const.dat` data file and set with the method `Flavour::set_mass()`, shadowing the values given in `Particle.dat`.

In contrast, the masses of the Z and W bosons are generated via spontaneous symmetry breaking and can be calculated to leading order, i.e. without quantum corrections, with the help of the coupling constants $\alpha_{\rm QED}$ and $\sin\theta_W$ and the Higgs vacuum expectation value (vev). In `AMEGIC++`, an option is provided to calculate these masses and set them, thus overwriting their values given in `Particle.dat`.

Note that in the framework of the Standard Model, these classes seem somewhat over-engineered, but for later extensions to models beyond the Standard Model spectra should be calculable in dependence to some set of parameters which might or might not be different from the sheer listing of particle masses. As an example take the Minimal Supersymmetric Standard Model, where — neglecting quantum corrections — the masses of the partner particles in general depend on the mass parameters of the Standard Model particles plus some symmetry breaking terms.

4.2.5 The couplings

`Couplings_EW` reads in the electroweak coupling constants and maintains the calculation for the running of $\alpha_{\rm QED}$:

1. `Init()` reads in the coupling constants of the electroweak model, i.e. $\alpha_{\rm QED}$ and $\sin\theta_W$ at the Z pole, the Higgs vacuum expectation value and the four parameters of Wolfenstein's parameterization of the CKM matrix. Accordingly, these values are set and the CKM matrix is generated. In order to calculate the running of $\alpha_{\rm QED}$ all thresholds are determined with `Thresholds()`. Then, the coupling constant as well as the β-function of the leading order running which depends on the number of active charged fermions is calculated via `Init_AQED()`.

2. `Thresholds()` determines all particle thresholds which contribute to the running of $\alpha_{\rm QED}$, i.e. they have to have an electric charge.

3. `Init_AQED()` determines $\alpha_{\rm QED}$ at the scales introduced by the different particle thresholds. In order to obtain an efficient evaluation of the running coupling constant, the β-function is precalculated as well.

4. aqed() returns α_{QED} at a given scale.

5. VEV() returns the Higgs vacuum expectation value.

6. SinThetaW() returns the $\sin\theta_W$.

7. CosThetaW() returns the $\cos\theta_W$.

8. CKM() returns the specified entry of the CKM matrix.

Note that the running of electroweak parameters other than α_{QED} has not been implemented so far. Couplings_QCD reads in the strong coupling α_{QCD} and calculates its running:

1. Init() reads in the only coupling constant of the strong interaction, α_{QCD} at the Z pole. Similarly to the procedure in Couplings_EW the thresholds as well as the coupling constants at these thresholds are determined using the methods Thresholds() and Init_As(), respectively.

2. Thresholds() specifies the thresholds for all strong interacting particles.

3. Init_As() calculates the value of the strong coupling at the different threshold scales as well as the β-function in order to perform a one loop running of the coupling.

4. as2() returns α_{QCD} at a given scale.

4.3 The amplitude

In this section we describe the generation of Feynman diagrams, their translation into the helicity amplitudes and their calculation. Accordingly, we start with the basic handling of amplitudes in section 4.3.1, then dwell on the generation of the Feynman diagrams in section 4.3.2 and on their representation within a Single_Amplitude in section 4.3.3. In this section the translation into helicity amplitudes is performed as well. We end with the description of the tools for the calculation of the amplitude in section 4.3.4 as well as the Color structure and Coulomb factors in section 4.3.5. A short outline of the involved classes and structures can be found in table 10, whereas a pictorial overview is given in figure 20.

4.3.1 Handling of amplitudes

The organization of the generation as well as the calculation of a Feynman amplitude is governed by the class Amplitude_Handler:

1. The Constructor governs the whole initialization procedure, i.e. the generation of Feynman diagrams and their translation into helicity amplitudes in the following way:

   - All Feynman diagrams are generated with the construction of an Amplitude_Generator object and the call to its method Matching(). This method constructs a list of Single_Amplitudes, each of them representing an individual Feynman diagram through a list of linked — and filled — Points.

Class/Struct	Purpose
Amplitude_Handler	Handles all amplitudes, i.e. the sum of all Feynman diagrams.
Amplitude_Generator	Generates all amplitudes, i.e. matches the vertices onto the topologies.
struct Pre_Amplitude	Is a `Point` list.
Single_Amplitude	Handles one Amplitude, i.e. Feynman diagram.
Zfunc_Generator	Translates the Feynman diagrams into helicity amplitudes.
Zfunc	Is a structure for storing one Z-function.
Prop_Generator	Finds and labels all propagators.
Pfunc	Is a structure for storing one propagator.
Color_Generator	Determines the color structure.
Cfunc	Is a structure for storing one color matrix.
Building_Blocks	The different building blocks of a helicity amplitude.
Mathematica_Interface	Interfaces to Mathematica, calculating some building blocks.
Composite_Zfuncs	Composed helicity amplitudes.
Elementary_Zfuncs	All elementary helicity amplitudes, i.e. Z, X- and Y functions.
Basic_Sfuncs	The S functions.
Momfunc	Is a structure which stores a four-momentum for incoming and outgoing particles as well as for propagators.
Color	Calulates the sum of color matrices.
Coulomb	Calculates Coulomb corrections for W-pair creation.

Table 10: All classes and structures which are used for the determination and calculation of a Feynman amplitude.

- A loop over all amplitudes transforms them into the helicity amplitudes with `Single_Amplitude::Zprojecting()`. The couplings are filled into the `String_Handler` by `Single_Amplitude::Fill_Coupling()`.
- Now, the dummy numbers for the polarization vectors can be replaced by the proper number within the Z-functions by calling `Polarization::Replace_Number()`.
- All the S-, η- and μ-functions depending on the four-momenta, including the propagators, are calculated within `Basic_Sfuncs`. Having at hand all Z-functions and the list of propagators `Pfunc`, they can be initialized with `Basic_Sfuncs::Build_Momlist()`.
- Last but not least the matrix composed of colour factors for each pair of amplitudes is calculated by constructing the `Color` object.

2. `Get_Graph()` returns the `Point` list of a Feynman diagram for a given number.

3. `Fill_Coupling()` performs a loop over all `Single_Amplitudes` and fills the couplings into the `String_Handler` via `Single_Amplitude::Fill_Coupling()`.

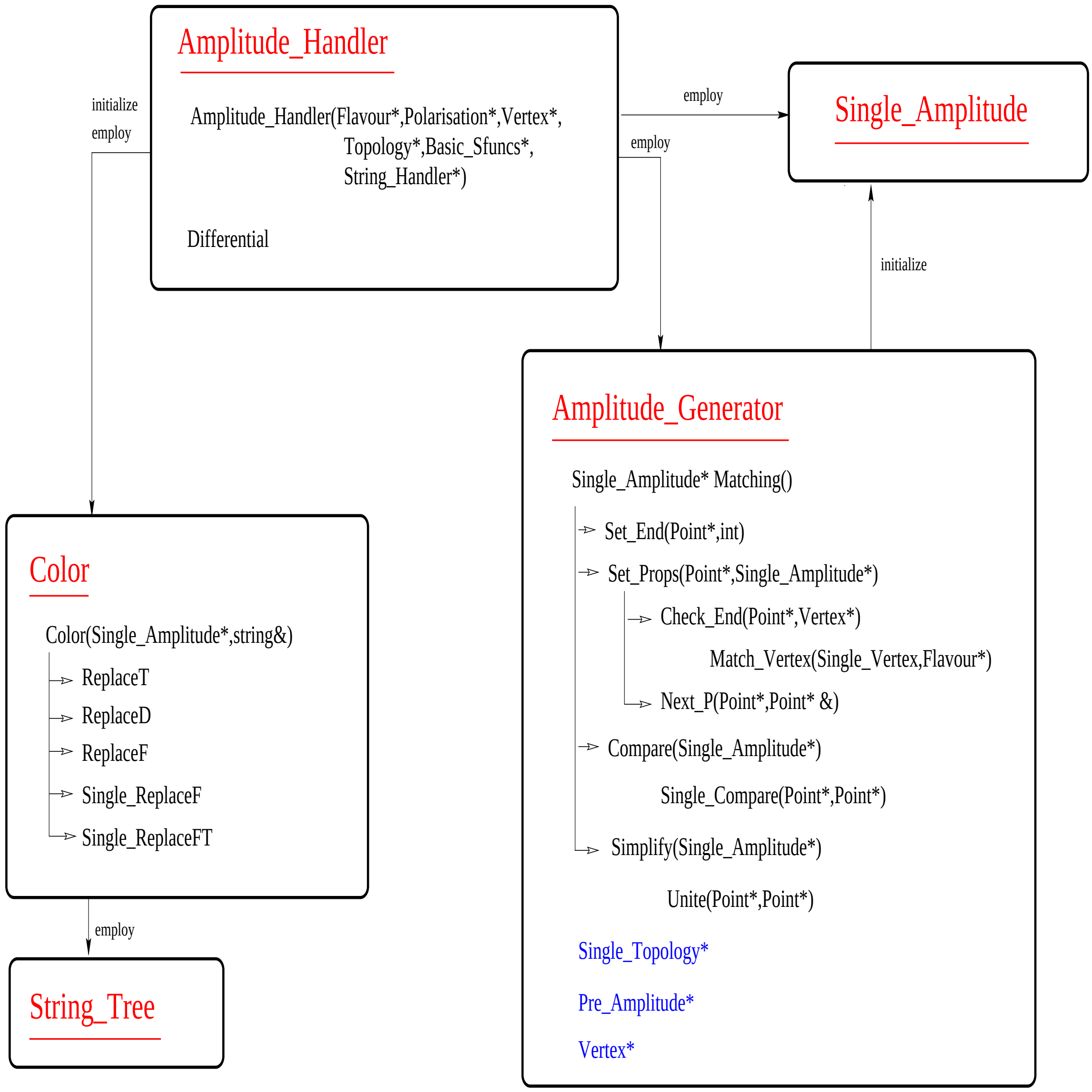


Figure 20: The interplay between the organizing class `Amplitude_Handler` and the classes for generating, storing and calculating a single amplitude (Feynman diagram).

4. `Get_Graph_Number()` returns the number of graphs.

5. `Get_Pointlist()` returns the `Point` list of a Feynman diagram for a given number. Note that this method considers the full list of graphs, not the shortened version of the `Single_Amplitude` list, where some graphs are united which differ only by a photon, Z or Higgs boson propagator.

6. `Get_PointlistNumber()` returns the number of the full `Point` list.

7. `differential(int*,int)` calculates the differential cross section by using the standard calculation methods, i.e. no character strings. The arguments of the method are a

list of signs for the different helicities of incoming and outgoing particles, whereas the second argument corresponds to the counting number of the current helicity combination. A loop over all amplitudes and a call to `Single_Amplitude::Zvalue()` fills the complex value of one diagram for a specific helicity combination into an array. At the end, taking into account the elements of the color matrix (`Color::Mij()`), all diagrams are summed up. Sometimes, single diagrams yield zero depending on the structure of the couplings. This is taken care of by storing the value of each diagram and dropping those which yield no result at all after a user defined number of calls, typically one thousand.

8. The second `differential(int)` method differs from the other one by two points. First, the amplitude is calculated using the already generated character strings for the evaluation of each diagram via `String_Handler::Zvalue()`. Accordingly, only one argument is needed for this method, i.e. the current counting number of the helicity combination. Second, since these character strings are already shortened, i.e. zero parts are dropped, no special care has to be taken for zero diagrams.

4.3.2 Generating the Feynman diagrams — `Amplitude_Generator`

The structure `Pre_Amplitude` stores a list of points. Therefore it can be regarded as a predecessor of a proper `Single_Amplitude`. In fact, until this `Pre_Amplitude` has not passed certain tests it cannot become a full amplitude. The generation of these `Pre_Amplitudes` as well as the handling of the full amplitudes is governed by the class `Amplitude_Generator` by means of the following methods:

1. The `Constructor()` obtains the topology which fits the actual number of legs via `Topology::Get()` and generates an empty list of `Pre_Amplitudes`.

2. `Matching()` is the main method within this class. It constructs the list of `Single_Amplitudes` and simplifies them. For the first step all permutations of the incoming and outgoing particles are constructed. This is achieved by a loop over loops technique (cf. appendix C), where the permutations are generated and then have to obey certain constraints, listed in section 3.1. Having passed these constraints a permutation of incoming and outgoing particles can be matched on all possible topologies. First of all, these particles are set to the endpoints of a certain topology with `Set_End()`. After this, all possible propagators in between are tested with `Set_Props()` and eventually diagrams are generated. At the very end of the loop over loops all diagrams are stored into a list of `Single_Amplitudes`. However, similar diagrams could still occur and have to be dropped. This is achieved with the method `Compare()`. Having at hand the full list of distinguishable diagrams, they are stored (`Save_Pointlist()`) for the later use in the construction of `Channels` employed during phase space integration. Now the list of diagrams can be shortened by unifying those which differ only in a photon, Z or Higgs boson propagator with the method `Simplify()`.

3. `Set_End()` recursively sets all endpoints of a topology onto a particle permutation, i.e. the particle number, its flavour and the `b` flag which marks incoming and outgoing particles. Propagators are labeled with numbers beginning with 100.

4. `Set_Props()` performs the matching procedure for one topology and one combination of endpoints. Basically, it tries to find every possible flavour combination for the inner propagators in the following way:

   At first, the raw topology with the endpoints already set is copied into a `Pre_Amplitude`. Then a loop over all `Pre_Amplitudes` is performed. Within the loop, every time, an internal line is equipped with a flavour, i.e. each time a new propagator is tested, a new `Pre_Amplitude` is created and appended. Consequently, the loop over `Pre_Amplitudes` ends, if no more propagators can be tested. The test is performed along the following steps:

   - The next free propagator in the actual `Pre_Amplitude`, i.e. a `Point` where the flavour is not set already, is selected via `Next_P()`. If no free `Point` can be found, the loop advances to the next `Pre_Amplitude`.
   - A loop over all possible vertices is performed. Each vertex is compared with a flavour combination which is manipulated in the way described in section 3.1. This is done with `Match_Vertex()`.
   - If the comparison was successful, the flavours of the outgoing particles are set onto this vertex. If they are connected with endpoints, the left and the right `Points` are checked by (`Check_End()`).
   - Having passed all these tests, the actual `Pre_Amplitude` is copied. It is attached at the end of the list of `Pre_Amplitudes` and the appropriate vertex is changed in the copy. This method ensures that all other vertices can be tested as well for this particular `Point`.
   - If the loop over the vertices has finished, the loop over the `Pre_Amplitudes` proceeds to the next amplitude and the actual one is switched off. Therefore, every new `Point` which has matched onto a vertex, yields a new amplitude. In this way, only amplitudes which are fully filled, reach the end of the loop. If no new amplitude exists, the loop terminates.

   In this fashion, a number of pre-amplitudes has been generated which will be subjected to some simple tests in order to avoid double counting of Feynman diagrams:

   - If the chosen model is QCD (all QCD Feynman rules plus a coupling to electrons and positrons), only diagrams can survive which have just one electroweak (i.e. photon, Z boson) propagator. This is used for the generation of QCD final states in electron-positron annihilations.
   - In multiple boson vertices a certain ordering of the particles has to be demanded in order to avoid a double counting of, for instance, three gluon vertices.

Passing these tests the located Pre_Amplitude can be translated into a proper Feynman diagram, i.e. a Single_Amplitude. An additional minus sign for every exchange of two fermions is included and stored as well.

5. Next_P() searches recursively through the list of Points for the next undetermined propagator, i.e. for a Point with the Flavour not yet set.

6. Print_P() prints a Point list recursively.

7. In Match_Vertex() a given combination of Flavours is compared with a given Single_Vertex, i.e. all Flavours which are set will be compared and if they match, the unset Flavours as well as the coupling constants can be filled.

8. Check_End() checks for a given Point with already set flavour whether the linked left and right Points are already endpoints and, if this is the case, whether they match. For the matching part, a loop over all possible vertices is performed and every vertex is compared with a slightly modified flavour combination of the incoming and the two outgoing particles. The flavours are changed according to the rules presented in section 3.1. Having at hand the proper flavour combination it is checked against the actual vertex via Match_Vertex(). If the checks are passed, the couplings of the Point are set.

9. Compare() performs a comparison of all amplitudes. Since the derived list of Feynman diagrams could still have some double countings, this step is necessary. Within a double loop over all Single_Amplitudes the appropriate Point lists are compared with Single_Compare(), where one of them is marked for deletion in case they are equal. This is achieved by using the On switch of a Single_Amplitude. Finally, all marked amplitudes are deleted with the method Kill_Off().

10. Single_Compare() compares two Point lists recursively which means that in one recursion step always two Points are inspected. They are checked for equal Flavour and number. After this, the left and right Points are compared with a recursive call to Single_Compare(). However, if the regarded Point belongs to a triple gluon vertex, the left and right branch could be exchanged and compared again. Thus, all possible combinations of exchanging the different branches are considered.

11. Kill_Off() deletes all marked Single_Amplitudes which are switched off. A loop over all possible amplitudes and an appropriate reordering of the list of Single_Amplitudes is sufficient for this purpose.

12. Save_Pointlist() saves all Single_Amplitudes into a list of Point lists. This is necessary for the later use within the Channel_Generator for the phase space integration.

13. `Simplify()` compares all different amplitudes. If two of them differ only by a photon, Z or Higgs boson propagator they can be united and regarded as the same amplitude. Later on, during the calculation of the different pieces of helicity amplitudes, the contributions of the different propagators will be summed up. Therefore, always two `Point` lists are compared. In case they differ only be the regarded boson propagators, they are unified with the method `Unite()`.

14. `Unite()` unites two amplitudes which differ only by a photon, Z or Higgs boson propagator. Therefore, the arrays of couplings for the start and end `Points` of the propagator have to be enhanced to include the appropriate coupling of the other propagator types. The `Flavour` of the propagator is exchanged with a list of `Flavours` for all occurring flavour types.

4.3.3 A single amplitude

The `Single_Amplitude` is an implementation of a particular Feynman diagram. Therefore, it includes the `Point` list for the graph, a list of `Zfuncs` for the translation of the graph into helicity amplitudes, a list of propagators and a list of color matrices. The class governs the generation of all these lists and calculates the amplitude with the help of `Building_Blocks`, for a pictorial overview of the connection to other classes see figure 21. Furthermore, `Single_Amplitude` provides a number of methods for manipulating the generated lists.

1. The `Constructor()` initializes the list of `Points`. However, during the generation of this list in the `Amplitude_Generator` not much care has been taken to the correct direction of the spin flow for the first fermion line. Therefore, a number of propagators might still have the wrong spinor direction, i.e. they are considered to be particles instead of anti-particles or vice versa. In order to cure this problem, two methods (`Initial_Fermionline()` and `Reverse_Fermionline()`) are used in the following way:

   - If the first particle is an incoming fermion and not an anti-particle, the initial spinor line is marked with `Initial_Fermionline()`. Furthermore, the endpoint of this line is determined as well.
   - If the endpoint of this fermion line is an outgoing particle, the spin flow of this first spinor line is reversed beginning with the last particle in `Reverse_Fermionline()`.

   During these two steps a special flag in the `Point` list (t) is set. Later on, during the generation of the list of propagators, this flag determines the particle or anti-particle character of the appropriate propagator.

2. `Initial_Fermionline()` determines recursively the initial fermion (i.e. spinor) line and its endpoint. The pointer to the previous `Point` is set as well.

3. `Reverse_Fermionline()` starts from the endpoint of the first fermion line. The t flag is set equal to one in case the propagator is an anti-particle.

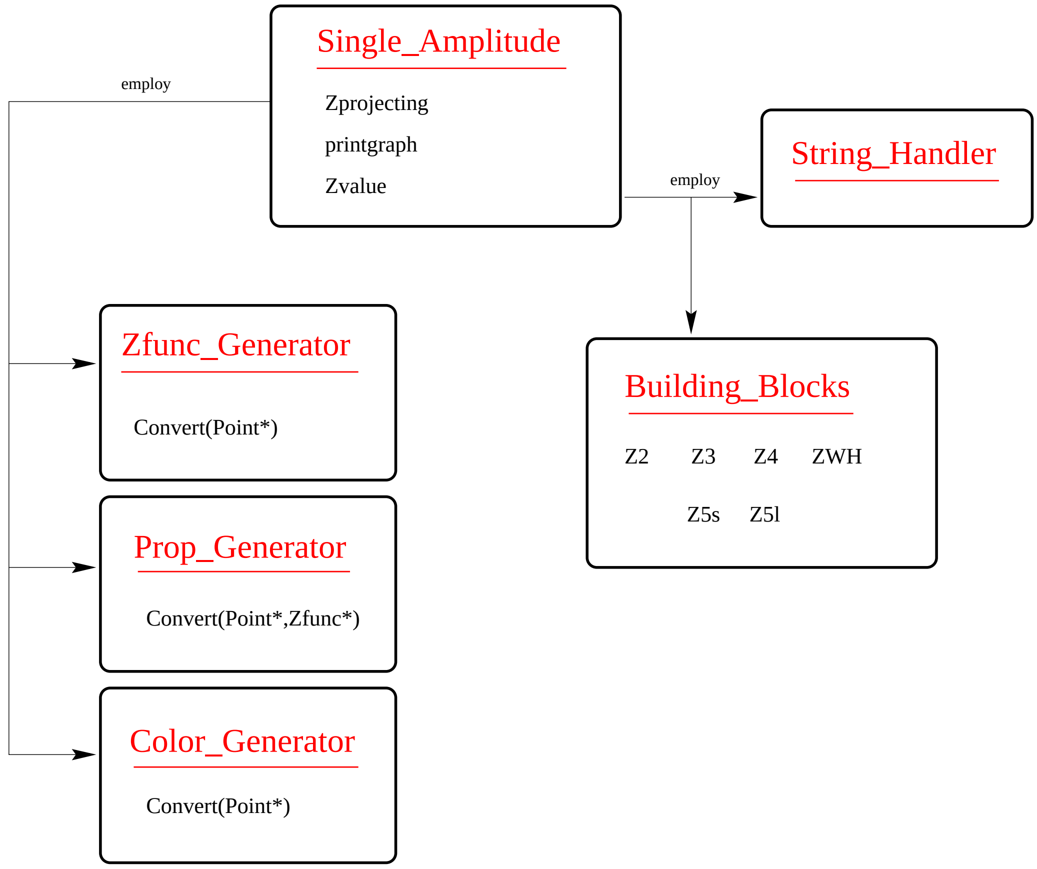


Figure 21: One Feynman diagram.

4. `Zprojecting()` translates the Feynman diagram into the helicity amplitude language in the following steps:

   - The colour structure of the Feynman diagram is determined using the `Color_Generator`. With `Color_Generator::Convert()` every vertex gains a colour matrix and `Color_Generator::C2string()` translates the whole term into a character string which will be used for later evaluation.
   - The various pieces of a helicity amplitude are generated within the `Zfunc_Generator`. Therefore, `Zfunc_Generator::Convert()` interprets the Feynman diagram, i.e. the list of `Points` in terms of helicity amplitudes and `Zfunc_Generator::Flip()` changes the sequence of pairs of barred and unbarred spinors, so that in each pair, barred spinors come before unbarred ones in accordance with the definition of the X-, Y- and Z-functions.
   - The `Prop_Generator` constructs a list of propagators, i.e. determines the momenta of the different propagators in terms of incoming and outgoing particles. With `Prop_Generator::Convert()` the list is generated, `Prop_Generator::Fill()` fills in the ingredients (dependencies on external particles) of every propagator and kills

the ones which are not used (`Prop_Generator::Kill()`). In this method, the list of `Zfuncs` is explored and every propagator which could not be found, is deleted.

Now, lists of `Zfuncs`, `Cfuncs` and `Pfuncs` for the helicity amplitudes, the color matrices and the list of propagators, respectively, are at hand.

5. `Fill_Coupling()` fills all couplings into the `String_Handler`. For this purpose, a loop over all `Zfuncs` is performed and every particular coupling is set via `String_Handler::Get_Cnumber()`.

6. `MPolconvert()` converts polarizations, i.e. replaces every number in the list of `Zfuncs` with a given new number for the polarization.

7. `Prop_Replace()` adds a new propagator to the list of `Pfuncs`. This method is commonly used for massive polarization vectors, where an external massive vector boson is treated like a propagator decaying into its polarizations.

8. `Zvalue()` calculates the helicity amplitude for one Feynman diagram:[5]

   - The maximum number of fermionic propagators (i.e. those which are switched on) is determined and their numbers are stored into an array.
   - If this number of propagators equals zero, only one `Zfunc` has to be calculated. The arguments of the Z function as well as their helicities are filled into a list of arguments via `Fill_Args()`. Then, the appropriate couplings are set via `Building_Blocks::Set_Arg_Couplings()`. Now the Z function can be calculated with `Single_Zvalue()`.
   - If any fermionic propagator exists, a loop over all possible states for every fermion propagator has to be performed. Since the number of propagators is not known in advance, a loop over loops technique has to be applied (cf. appendix C). In the external loop the sign of the propagators which correspond to a particle or anti-particle state, and the two helicity states have to be altered. Having at hand one such sample, the product of all Z functions for this amplitude has to be calculated using a loop over all `Zfuncs`. The evaluation is performed in the same way as described above for a single `Zfunc`. Now, the product of all Z's is multiplied with an extra mass term (`Mass_Terms()`) which corresponds to the real and virtual propagator mass, see eq. (2.13). The contributions for every set of propagator states is summed up and multiplied at the very end with all emerging fermion propagators (`Building_Blocks::Set_P()`) and with a factor of one half taking into account the sum over particles and anti-particles.

   Finally, the calculated value multiplied by the overall sign of the amplitude is returned.

[5]Note, that by construction, the boson propagators are already part of the Z-function.

9. `Fill_Args()` constructs the list of arguments for one Z function. For this purpose it uses the list of helicities for the incoming and outgoing particles as well as the list of propagators with their appropriate helicity states. A loop over the arguments of the Z function is performed and translated in the following way:
   - If the argument is a propagator, i.e. the number is larger than 99, a signed propagator number and the appropriate helicity is added to the list of arguments. The sign corresponds to a particle or an anti-particle state propagating.
   - If the argument is an incoming or outgoing particle, its number and helicity are taken over unchanged.
   - If the argument corresponds to a polarisation of a massless vector boson, its number is larger then the maximum number of particles and smaller than 20. In this case, the helicity of the polarisation is the same as the helicity of the appropriate massless vector boson and the number of the momentum is translated via subtracting 10.
   - The number of the polarisation for a massive vector boson is always between 20 and 99. These polarisations get a helicity of minus one by construction and the momentum number can be gained by subtracting 20.
10. `Single_Zvalue()` calculates one `Zfunc` using the various possibilities of the `Building_Blocks`.
11. `Mass_Terms()` calculates the mass term of eq. (2.13) for the decomposition of fermionic propagators using `Building_Blocks::Set_Mass()`.
12. `printgraph()` prints all lists for one Feynman diagram, i.e. the list of `Zfuncs`, `Pfuncs` and `Cfuncs`.
13. `Get_Pointlist()` returns the list of `Points`.
14. `Get_Clist()` returns the list of colour matrices, i.e. `Cfuncs`.
15. `Get_Plist()` returns the list of propagators, i.e. `Pfuncs`.

The structure `Zfunc` includes all informations which are necessary to calculate an arbitrary Z function, namely the type $(-1\dots 9)$, the list of arguments and the list of couplings, see table 11.

Last but not least every bosonic propagator which is involved in this Z function is stored as well in a special list. This will become necessary later on for the calculation of massive vector boson propagators.

The class `Zfunc_Generator` constructs the list of `Zfuncs`, i.e. it translates a Feynman diagram into the helicity language by decomposing it into a number of Z functions. This is one of the central parts of the program. Note that during the determination of the list of arguments for every Z function, a certain internal numbering scheme is applied, for details see table 12.

Type	#arg	#coupl	#prop	Description
-1	2	2	0	A scalar boson which is connected to two fermions.
0	4	12	0	An outgoing or incoming vector boson attached at a fermion line
			1	or two fermion lines connected via a vector boson.
1	6	10	3	A three vector boson vertex.
2	6	7	3	A two vector and one scalar boson vertex.
3	6	10	3	A one vector and two scalar boson vertex.
4	0	1	0	A three scalar boson vertex.
5	8	10	5	The four vector boson vertex.
9	10	13	7	The five vector boson vertex.

Table 11: All possible types of a `Zfunc` and their description as well as the number of arguments, couplings and propagators.

Particle type	Number
Incoming or outgoing	0–9
Fermion propagator	100–120
Boson propagator	200–
Polarization of a massless vector boson	Number of the boson plus 10.
Polarization of a massive vector boson	Number of the boson plus 20.
Dummy numbers for the calculation of the Color matrices	120–140

Table 12: The particle numbering scheme in the list of arguments for every Z function.

1. `Convert()` translates a given graph, i.e. a list of `Points` into the list of `Zfuncs` recursively. A new `Zfunc` will be produced, if the current particle is a fermion, a scalar boson or an incoming vector boson. Depending on the `Flavour` type of this particle, two pointers will be set. In case it is a fermion, one pointer is set to the outgoing boson and one to the outgoing fermion. If the particle is a boson, only the boson pointer is set, whereas the fermion pointer is set to zero. With the current `Point` and these two pointers as arguments the type of the `Zfunc` can be determined (`Determine_Zfunc()`) and the arguments can be filled in (`Fill_Zfunc()`). At the end two recursive calls to `Convert()`, with the left- and the right-hand `Point` as arguments, ensure the recursive examination of the whole graph.

2. `Determine_Zfunc()` determines the type of the `Zfunc` according to table 11, where the method `IsGaugeV()` is used to count the number of vector and scalar bosons at one vertex.

3. `IsGaugeV()` determines the number of vector and scalar bosons within one vertex.

4. `Fill_Zfunc()` specifies all arguments, couplings and propagators of one `Zfunc`. First of all, the sizes of all three lists are determined (see table 11). Now, the different lists

can be initialized and filled by calling the appropriate method. Note that each type of Zfunc has its own method (Zm1(), Z0(), etc.).

5. Set_In() sets the arguments and couplings for the incoming part of a Zfunc. If the fermionic pointer is set (see Convert()), the arguments are set on the current and the fermionic pointer, where the sequence is always anti-particles before particles. The two couplings will be filled with the couplings from the current pointer. Note that the sign of the graph gains an extra minus for a fermionic anti-particle propagator. The second case is addressed to an incoming boson. If it is scalar, both arguments are set on the number of this boson and the couplings are set to zero (this is only the dummy part of a Zfunc). For a vector boson the first argument is set on the number of the boson and the second is set to the number plus 20 or 11 for massive or massless vector bosons, respectively. Here, the couplings are set to unity.

6. Set_Out() sets the arguments, couplings and propagators for one outgoing end of a Zfunc. At the beginning, the propagator will be set onto the current boson. If the boson is not an outgoing one the arguments of the Zfunc are set onto the numbers of the left- and right-hand particle. The sign of the graph will be changed for a fermionic propagator on the left-hand side (i.e. an anti-particle propagator). Outgoing bosons will be managed in the same way like incoming bosons (see Set_In()), but with an reversed sequence of arguments.

7. Set_Scalar() sets the left- and right-handed couplings to a scalar boson within a Zfunc. This means that both arguments are set to the number of the scalar boson and the couplings are set to zero. The list of propagators is enhanced by the scalar boson propagator.

8. Get_Flav_Pos3() determines the position of the three electroweak vector bosons in the method Z1().

9. Zm1() sets the arguments with Set_In() or Set_Out() for an incoming or outgoing scalar boson, respectively.

10. Z0() specifies the arguments and couplings for a vector boson exchange between two fermion lines or an incoming or outgoing boson:

    - For an incoming or outgoing scalar boson the incoming part will be set on position 0 (Set_In()) and the outgoing part at position 1 (Set_Out()).
    - The same holds true for an incoming vector boson.
    - In the case of outgoing vector bosons or the exchange of a vector boson between two fermion lines the outgoing part is set on position 0 and the incomings are set on position 1 with Set_Out() and Set_In(), respectively.

    Note that the sequence of the arguments plays an important role during the calculation of the Z function. However, multiple propagators (γ, Z, Higgs-boson) will be considered by enhancing the list of couplings of the Zfunc.

Method	Incoming	Left	Right	Condition
`Z1()`	2	0	1	For a three gluon vertex.
	2	0	1	For $\gamma/Z, W, W$ vertex.
	0	2	1	For $W, \gamma/Z, W$ vertex.
	0	1	2	For $W, W, \gamma/Z$ vertex.
`Z2()`	2	0	1	For $H, Z/W, Z/W$ vertex.
	0	2	1	For $Z/W, H, Z/W$ vertex.
	0	1	2	For $Z/W, Z/W, H$ vertex.
`Z3()`	2	0	1	For V, S, S vertex.
	0	2	1	For S, V, S vertex.
	1	0	2	For S, S, V vertex.
`Z4()`				For S, S, S vertex (is only a coupling).

Table 13: The positions in the list of arguments for the different three boson vertex types and constellations.

Method	I	L	LL	LR	R	RL	RR	Condition
`Z5()`	0		3	2	1			For $\gamma/Z; W; \gamma/Z; W$ vertex.
	1		3	0	2			For $W; W; \gamma/Z; \gamma/Z$ vertex.
	0	1				2	3	For $\gamma/Z; W; \gamma/Z; W$ vertex.
	1	0				2	3	For $W; \gamma/Z; \gamma/Z; W$ vertex.
	0	1				2	3	For $G; G; G; G$ vertex.

Table 14: The positions in the list of arguments for the four vector boson vertex with different constellations. Note that I stands for incoming, L and R for left and right, respectively.

11. The methods `Z{1,2,3,4}()` correspond to a three boson vertex, where the positions within the list of arguments depend on the appropriate vertex, for details see table 13. Note that all incoming bosons are placed with `Set_In()`, outgoing bosons are set with `Set_Out()` or `Set_Scalar()` for vector or scalar bosons, respectively. However, multiple propagators can occur in the case of vector bosons, too. Then, the list of couplings will be enhanced by the extra couplings.

12. `Z5()` sets the arguments for the four vector boson vertex, see table 14. Apart from that, all other options are the same like in the three boson vertex case.

13. `Z9()` considers the calculation of the five vector boson vertex. Note that in the current version only the five gluon vertex is used, i.e. no five electroweak gauge boson vertices are taken implemented yet. This is part of future work.

14. `Flip()` takes care for the correct order of particle and anti-particles in the list of arguments in the `Zfuncs`, i.e. the sequence of spinors. Here, anti-spinors always have to stand before spinors in the appropriate two-particle blocks of the `Zfuncs`. Therefore a loop over all `Zfuncs` is performed and the arguments of every function are examined. Now, a change will occur in case an incoming fermion or an outgoing anti-fermion is placed before its partner. The same change will be enforced, if the second particle is an incoming anti-fermion or an outgoing fermion. Both checks are necessary, since one of the arguments could be a propagator, where the spinor or anti-spinor

property is not fixed before. If a change should be performed and the partner of the examined particle is a propagator, not only the current `Zfunc` but also a second one will be altered. This is due to the fact that a fermionic propagator always emerges in two `Zfuncs`. Now, if the propagator stands on the first position after the change, the second occurrence of the propagator in another `Zfunc` has to be as a second argument. Therefore, the other `Zfunc` will be searched and changed accordingly. However, this second occurrence of the propagator can have a propagator as a partner as well. Consequently, the change of the position for this partner results in another change for its second appearance and so on. In this way, the whole fermion line will be flipped.

15. `Get()` returns the list of `Zfuncs`.

The structure `Pfunc` contains all informations to characterize a propagator. The necessary arguments are the number of the propagator and all those incoming and outgoing momenta which sum up to its momentum. A `Flavour` gives the type of the propagator and a complex number stores the calculated value. Last but not least an on switch allows the dropping of individual propagators.

The class `Prop_Generator` generates the list of propagators, i.e. the `Pfuncs`, and calculates their complex value for one individual graph:

1. `Convert()` translates a topology, i.e. a list of `Points` into a list of propagators (`Pfuncs`). A propagator is characterized through a number which is larger than 99 (this corresponds directly to our numbering scheme, see table 12). Now, a recursive examination of the whole tree yields all possible propagators. Note that some of the propagators are combined for different boson types (i.e. photon, Z, and Higgs boson). Since different boson types yield different propagator values, a new propagator will be generated for every boson. For every propagator a `Pfunc` is created and attached to the list. The first argument of the `Pfunc` is the number of the appropriate propagator and the two next arguments correspond to the numbers of the left- and right-handed particles in the graph. At the end of the method two recursive calls to `Convert()` will be performed.

2. `Fill()` completes the list of arguments for the propagators. In `Convert()` every propagator is filled into a `Pfunc`. The arguments of this `Pfunc` are the flavour of the propagator and the left- and right-handed particles. However, in order to calculate the four-momenta of the propagators later on, the dependence on the incoming and outgoing particles is needed (since only the four-momenta of these particles are available). The task of this method is to replace the dependence of the propagator on the left- and right-handed particle by the dependence on all (incoming and outgoing) particles which are attached at the same branch. Therefore, a loop over all `Pfuncs` is enclosed by a loop which ends, when all dependencies are filled in. Now, every `Pfunc` is examined. If a propagator is found in the arguments, the number of this propagator is replaced by the list of dependencies for this propagator itself. These

replacements are performed until every propagator has a list of dependencies which includes only the numbers of incoming and outgoing particles.

3. `Kill()` deletes all propagators which do not appear as an argument in the list of `Zfuncs`. A loop over all propagators and Z functions searches for the number of the current propagator in the list of arguments of the current Z function. If a propagator could not be found, it will be switched off.

4. `Calculate()` evaluates the complex value of all propagators. Consequently, a loop over the list of `Pfuncs` is performed. Within the loop first of all, the four-momentum of the current propagator has to be determined. This can be achieved by adding up all the depending four-momenta of incoming and outgoing particles, i.e. all four-momenta which are attached at the same branch of the tree as the propagator. Note that the four-momenta of the incoming particles are multiplied with a -1. Now, the complex value of the propagator can be calculated using the mass and the width of the current particle. According to our Feynman rule conventions every fermionic and scalar particle is multiplied with an extra complex unit and the vector bosons are provided with minus a complex unit.

5. `Get()` returns the list of `Pfuncs`.

The structure `Cfunc` represents an element of the color matrices, i.e. a generator of the SU(3) color group (T^a_{bc}) or a structure constant of the group (f_{abc}). They are indicated by different types within `Cfunc`, where a 10 represents a delta function, a 0 corresponds to a generator of the fundamental representation and the number 1 is connected with a structure constant. The appropriate labels (abc) are saved in a list of arguments.

The `Color_Generator` generates a list of these `Cfuncs` which belong to every graph. A string representation will be created as well, in order to simplify the calculation of the color factors later on:

1. `Convert()` generates the list of `Cfuncs`. Therefore, the whole topology of a graph is searched recursively and every vertex yields a new color matrix `Cfunc` depending on the incoming and the two outgoing particles of this vertex (see table 15).

   At the end two recursive calls to `Convert()` (with the left- and right-hand side `Point`) ensure that every vertex in the topology is translated into a `Cfunc`.

2. `Kill()` performes a first simplification of the color structure, i.e. all delta functions which involve propagators, are deleted and the appropriate replacements are done within the list of `Cfuncs`. Accordingly, a loop over all `Cfuncs` is performed and every delta function (i.e. type equals 10) is examined. If one of the two arguments is a propagator, all emerging numbers of this propagator will be replaced by the second argument in all `Cfuncs`. At the end these `Cfuncs` which correspond to a delta function with propagator indices only, are deleted. Note that delta functions remain which have incoming or outgoing particles in the argument.

Incoming	Left outgoing	Right outgoing	Type	arg0	arg1	arg2	Condition
Quark	Boson	Quark	0/10	L	R	I	Incoming anti-quark.
				L	I	R	Incoming quark.
Quark	Quark	Boson	0/10	R	L	I	Incoming anti-quark.
				R	I	L	Incoming quark.
Boson	Quark	Quark	0/10	I	L	R	
Boson	Boson	Boson	1	I	L	R	

Table 15: In this table the different allocations for the arguments of a `Cfunc` are shown. The left, right and the incoming particle are denoted as `L`, `R`, and `I`, respectively. Note that the type depends on the nature of the boson. In case it is a gluon, the type is 0 and for all other bosons the type is 10.

3. `C2string()` translates the list of `Cfuncs` into a character string. During the generation of the string, every type of `Cfunc` will be translated accordingly, i.e. type 0 into a $T[A, b, c]$, type 1 into a $F[A, B, C]$ and type 10 into a $D[a, b]$ function. Note that every upper case letter lies in the range of $1, \ldots, 8$ and the lower case letters in the range of $1, \ldots, 3$. Furthermore, some care has to be taken that propagators which appear in different `Cfuncs`, gain the same letter. However, at this stage not only the color string will be created, but also its complex conjugate. Here, every propagator has to get a totally new letter with respect to the propagators in the original string, since these letters are only inner summation indices. Consequently, the color factor can now be calculated by just merging those two strings and no special care has to be taken for inner summation indices any more. This simplifies the calculation significantly later on (see the class `Color`).

4. `Get()` returns the list of `Cfuncs`.

4.3.4 Calculating the amplitude

The calculation of the amplitudes is organized on different abstraction levels. On the highest level stands the class `Building_Blocks` which evaluates the different composed Z functions and will be called from within `Single_Amplitude`. The building blocks themselves rely on the `Composite_Zfuncs` and the `Mathematica_Interfaces`, where the latter one uses the methods of the `Composite_Zfuncs` as well. Strictly speaking, these composed Z functions translate a simple notation of the different Z functions into a call with all arguments, where the methods of the class `Elementary_Zfuncs` will be used. These methods calculate all the Z,X,Y, etc. functions with the help of a set of basic spinor products and normalizations which are established in the class `Basic_Sfuncs`. This is the lowest level class which directly uses the four-momenta of the incoming and outgoing particles.

The `Building_Blocks` are the basic elements of a calculation. They calculate one single Z function and are called from within `Single_Amplitude::Zvalue()`:

1. With the `Constructor()` of this class a new object of `Composite_Zfuncs` is initialized. Together with the `Mathematica_Interfaces` they form the basic calculational tools of this class.

2. `Reset()` resets the `Composite_Zfuncs` object (via the appropriate `Reset()` method).

3. `Set_Arg_Coupl()` is used to set the list of arguments and couplings within this class and the `Composite_Zfuncs` object.

4. `Set_P()` calls `Composite_Zfuncs::Set_P()` and calculates the `Kabbala` value of the given propagator (`Pfunc`).

5. `Set_Mass()` calculates the mass term which occurs during the decomposition of the fermionic propagator into products of spinors (via `Composite_Zfuncs::Set_Mass()`).

6. `Z2()` calculates the `Kabbala` value of the original Z function. Here, the different contributions of multiple propagators (photon, Z and Higgs boson) are summed up as well. The Z function is calculated via `Composite_Zfuncs::Z_Z()` and the propagator via `Composite_Zfuncs::Z_P()`. Note that the contribution of a scalar boson exchange is composed of the product of two Y functions (`Composite_Zfuncs::Z_Y()`). The last missing piece is the mass term which appears in the numerator of the propagator for massive vector boson exchanges ($p^\mu p^\nu/M^2$). This term can be composed out of two X functions (`Composite_Zfuncs::Z_X()`) and a constant which is directly handled using the method `String_Generator()::Get_Enumber()`. Note that this method translates a complex number into a `Kabbala` object and saves the value in the list of `Zfuncs` (see section 4.4).

7. All other methods calculate an appropriate n boson vertex, usually with the help of a `Mathematica_Interface()`. A new object will be created the first time this method is called. The result is a product of the list of the inner propagators (gained via `Composite_Zfuncs::Z_P()`), the value of the vertex (`Mathematica_Interface()::Calculate()`) and the coupling constant (`String_Handler::Get_Enumber()`). A further complication can arise when multiple propagators occur (i.e. a photon and a Z boson). Then the contributions will be simply summed up. If more than three vector bosons appear in one vertex, a sum of the multiple iterated three vector boson vertices and a quadruple vector boson vertex has to be performed. This is due to our handling of quadruple vector boson vertices. However, a detailed list of every method can be found in table 16.

The `Mathematica_Interface` can be used to implement algebraic expressions which are produced by `Mathematica`. Here, the output of the appropriate `Mathematica` session will be translated into a proper `C++` file and linked to the object. This process can be done in a semi-automatic way. However, usually all necessary calculations have been already performed and the linked `C++` files are ready to use:

1. The `Constructor()` can be used with two different options. The first one which is the usual option, is used for the calculation of already translated methods. An identity string indicates the method to be used. In the second option a `Mathematica` file will be read in and translated into a `C++` file. First of all, the file will be read in and casted into a string. The method `Shorten()` deletes all occurrences of p and q characters

Method	Purpose
Z3V()	Three vector boson vertex,
	sums up multiple propagators with `Z31`.
Z31()	Three vector boson vertex.
ZVVS()	Two vector, one scalar boson vertex.
ZSSV()	Two scalar, one vector boson vertex,
	sums up multiple propagators with `ZSSV1()`.
ZSSV1()	Two scalar, one vector boson vertex.
ZSSS()	Three scalar boson vertex, simply a coupling.
Z4V()	Four vector boson vertex.
Z5V()	Five vector boson vertex.

Table 16: A selected list of methods for the calculation of an n boson vertex.

which are necessary in the `Mathematica` file, but are now obsolete. With `C_Output()` the string will now be translated into a `C++` file. All other parts, i.e. the linkage of this new method and the calling sequence have to be included by hand, a sample could be found in appendix B.

2. `C_Output()` transforms an algebraic expression in the form of a character string into a `C++` method of this class. Therefore a new file will be created and the output of the string expression is performed in the same way as the output of the libraries, see section 4.4, i.e. a maximum number of characters per line will be enforced.

3. `Shorten()` uses `Kill_String()` to delete all p and q characters

4. `Kill_String()` kills the occurrence of a given character in a string.

5. `Calculate()` is the steering routine of the calculational part of this class. According to the given string identity, this routine choses the appropriate method for the evaluation.

6. The methods `Z()`, `X()`, `S()`, `Y()` and `M()` calculate an elementary Z, X, S, or Y function or a mass term, respectively. The appropriate methods `Z_Z()`, `Z_X()`, `Z_S()`, `Z_Y()` and `Z_M()` of the class `Composite_Zfuncs` are used accordingly.

7. All other methods calculate a multiple boson vertex directly, see table 17.

The class `Composite_Zfuncs` translates a simplified form of a call to the elementary Z functions (`Elementary_Zfuncs`) into a full calling sequence. Therefore this class mainly exists for a better readability of the `Building_Blocks` and the `Mathematica_Interface`:

1. The `Constructor()` creates a new `Elementary_Zfuncs` object.

2. `Reset()` sets the pointers of the arguments and the couplings of a Z function to zero.

3. `Set_Arg_Coupl()` sets the arguments and couplings of a Z function.

4. `Set_P()` calls the method `Elementary_Zfuncs::P()` in order to calculate the value of a propagator.

Method	Purpose
vGGG()	Three vector boson vertex without any mass terms.
vZWW()	Three vector boson vertex including all mass terms.
vVVS()	Two vector one scalar boson vertex.
vSSV()	Two scalar one vector boson vertex.
vGGGG_it()	Two iterated three boson vertices as a part of the four vector boson vertex, masses are neglected.
vGGGG_q()	Quadruple vector boson vertex, masses are neglected.
vZZWW_it()	Like `vGGGG_it()`, but including all mass terms.
vZZWW_q()	Like `vGGGG_q()`, but including all mass terms.
v5G_it()	Three iterated three boson vertices as part of the five vector boson vertex, masses are neglected.
v5G_itq()	One triple plus one quadruple vector boson vertex, neglecting masses.

Table 17: A selected list of methods for the direct calculation of a part of a multiple boson vertex.

5. `Set_Mass()` calls the method `Elementary_Zfuncs::Mass_Term()` in order to evaluate the extra mass term of a decomposed fermionic propagator, see eq. (2.13).

6. `Z_P()` calculates the product of propagators for a Z function. Therefore, a list of propagator numbers is used to extract all propagators taking part in the current Z function. Their `Kabbala` values will be multiplied up and returned.

7. `Z_X()` calculates an X function and translates a simple combination of arguments into the full list and calls `Elementary_Zfuncs::X()`. Note that this method takes into account a sign convention which permits the proper handling of the four-momentum current.

8. `Z_S()` works in the same way like `Z_X()`, but calls `Elementary_Zfuncs::S()`.

9. `Z_Z()` translates the simple notation of an elementary Z function into the full call of `Elementary_Zfuncs::Z()`.

10. `Z_Y()` works like `Z_Z()` with a slight difference. Here, it can happen that both couplings are zero. In this case the value 1 will be returned.

11. `Z_M()` calls directly `Elementary_Zfuncs::M()` without any translations.

The `Elementary_Zfuncs` form the last step on the ladder to the calculation of the different Z functions. In this class they will be finally calculated and the `String_Generator` is filled with the different Z functions. At least two methods exist for each function, one for building the string part and the second to calculate the appropriate Z function. Sometimes different methods exist for different calling sequences within the string libraries.

1. The methods `Z()`, `Y()`, `X()` and `S()` calculate an elementary Z, Y, X function or a scalar product of two four-momenta and provide its value for the `String_Generator()`. Therefore all arguments and couplings will be filled into one list. Now, all propagators

which have numbers larger than 99 will be mapped onto real four-momenta (i.e. with numbers smaller than 100) via `Map()`. Then, the value of the function can be calculated via the appropriate `{Z,Y,X,S}calc()` and translated into a `Kabbala` type value with `String_Generator::Get_{Z,Y,X,S}number()`. Note that in case a string is to be built, the function with all arguments is stored within the `String_Generator`.

2. The methods `Zcalc()`, `Ycalc()`, `Xcalc()` and `Scalc()` calculate an elementary Y, X, Z function or a scalar product according to tables 1, 2 and 3, respectively.

3. `P()` calculates the value of a propagator and fills it into the `String_Generator`. First of all, the number of the given propagator will be determined via `Basic_Sfuncs::Get_Mom_Number()`. Then, the value is calculated with `Pcalc(Flavour fl,...)` and translated into a `Kabbala` value (`String_Generator::Get_Pnumber()`).

4. `Pcalc(int fl,...)` translates the integer valued kf-code into a proper `Flavour()` and calls `Pcalc(Flavour fl,...)`.

5. `Pcalc(Flavour fl,...)` calculates the value of a propagator using `Propagator()`.

6. `Propagator()` calculates the value of a propagator for a given four-momentum squared and the `Flavour`. Note that different types of the `Flavour` gain different extra factors, i.e. fermionic and scalar propagators yield an extra factor of i and vector boson propagators get a $-i$.

7. `Mass_Term()` generates the extra mass term which occurs during the decomposition of the fermionic propagator into spinor products. The first step is to determine the `Flavour` of the propagator from the given number. Then, the mass term is calculated with `Mass_Term_Calc(..., Flavour fl)` and it is filled into the `String_Generator` (via `Get_Massnumber()`).

8. `Mass_Term_Calc(..., int fl)` translates the given kf-code into a proper `Flavour` and calls `Mass_Term_Calc(..., Flavour fl)`.

9. `Mass_Term_Calc(..., Flavour fl)` calculates the mass term.

10. `M()` determines the mass for a given particle number in case it is a propagator. This method is used for the calculation of the mass terms in massive vector boson propagators. The term $1/M^2$ is then filled into the `String_Generator()` as a complex number (`Get_Enumber()`).

11. `Map()` translates a given propagator number (> 99) into a proper four-momentum number. Therefore the list of `Pfuncs` is searched for the given number. Now, the new number is determined from the appropriate `Pfunc` using `Basic_Sfuncs::Get_Mom_Number()`.

We now turn our attention to the calculation of the basic functions which are necessary for the determination of every Z function.

The structure `Momfunc` represents a four-momentum of a particle. Since in this structure not only incoming and outgoing momenta, but also the four-momenta of the propagators are stored, a list of arguments is available as well (compare `Pfunc`).

The class `Basic_Sfuncs` generates and translates a list of four-momenta, i.e. `Momfuncs`, into the basic functions of the helicity formalism, see eq. (2.23) for the S functions and eq. (2.25) for the definition of the μ and η functions. With these tools, all other Z functions can be calculated:

1. The `Constructor()` initializes the list of `Momfuncs` with the call to `Initialize_Momlist()`.

2. `Initialize()` constructs all necessary lists, i.e. for the μ-, η-, S_0- and S_1-functions which correspond to the $S(+,\cdot)$ and $S(-,\cdot)$ functions. Furthermore, a list of masses for the different particles is created which is used later on for the determination of the μ functions.

3. `Initialize_Momlist()` constructs the list of `Momfuncs` for the incoming and outgoing particles. Since at this stage no propagators will be initialized, the list of arguments is filled only with the number of the particle. At the end the current number of `Momfuncs` is returned.

4. `Build_Momlist()` adds to the list of `Momfuncs` all possible propagators. A loop over the given list of `Pfuncs` ensures that all propagators will be covered. First of all, a check will be enforced, if the current propagator is already in the list with `Get_Mom_Number()`. This method returns the number of a given `Pfunc` in the list of `Momfuncs` or returns a -1 in case this propagator does not exist within the list. Now, a new `Momfunc` can be established, where the list of arguments will be taken over from the `Pfunc`. At the end, the new number of `Momfuncs` will be returned.

5. `Print_Momlist()` prints the current list of `Momfuncs` including the dependencies for every propagator type momentum.

6. `Get_Mom_Number()` returns for a given `Pfunc` the number of its propagator in the list of `Momfuncs`. In case this propagator does not exist, a -1 is returned. Accordingly, a loop over all existing `Momfuncs` is performed. Each list of arguments for the current `Momfunc` and the `Pfunc` are now compared. In case they are equal, the current number of the `Momfunc` is returned. Note that the comparison is sensitive to different sequences within the list of arguments.

7. `Calc_Momlist()` initializes the list of `Momfuncs`. This means that the `Momfuncs` for incoming and outgoing particles will be simply set and all propagators are calculated depending on the given four-momenta using the list of arguments and the sign of each particle (see `Sign()`).

8. `setS()` is the main method which is called from outside for every new combination of incoming and outgoing four-momenta. Here, all the different functions as well as all four-momenta for the propagators will be calculated in the following steps:

- First of all, the four-momenta of all propagators will be calculated with `Calc_Momlist()`.
- Now, within a loop over all `Momfuncs` all μ- and η-functions will be set. However, whereas the η-functions can be calculated without any ambiguities, the sign of the μ-functions depends on the particle or anti-particle character. In our notation a minus sign comes with:
  - All anti-particles which do not correspond to the first incoming particle and
  - the first incoming particle which is not an anti-particle.

  All other particles yield no sign. Note that the sign should have no influence for the propagator part of the `Momfuncs`.
- In the last step the S_0- and S_1-functions will be calculated using an outer and an inner loop over the `Momfuncs`, since these functions depend on two four-momenta. The different relations between the functions will be used in order to minimize their number to be evaluated.

9. `N()` returns the normalization factor for two given four-momenta. This method is used during the calculation of the normalization for the massless vector boson polarizations.

10. `Momentum()` returns the four-momentum for a given particle number by searching the list of `Momfuncs`.

11. The methods `S0()`, `S1()`, `mu()` and `eta()` return $S(+;p_1,p_2)$, $S(-;p_1,p_2)$, $\mu(p)$ and $\eta(p)$.

12. `Sign()` returns the sign of a momentum, i.e. a -1 for incoming and a $+1$ for outgoing particles.

13. `Flav()` returns the `Flavour` of the given particle.

4.3.5 Calculating the color matrix and the Coulomb factor

The class `Color` calculates all color factors from the list of `Cfuncs` and the appropriate strings of color matrices which have been established in the class `Color_Generator()`. Therefore, the list of `Cfuncs` will be first shortened and then the calculation is performed by manipulating the color matrices in form of a character string, the different replacement rules are listed in table 18.

1. The `Constructor()` calculates all elements of the color matrix or reads them in from a saved file in the following steps:

   - At first, all diagrams (the list of `Single_Amplitudes`) will be switched on. Later on, this switch will be used to discard a repeated calculation of the color factor for identical color structures.

Method	Replacement
`Single_ReplaceFT()`	$f^{abc}T^c = -i(T^aT^b - T^bT^a)$
`Single_ReplaceF()`	$f^{abc} = 2iT^a_{ij}(T^c_{jk}T^b_{ki} - T^b_{jk}T^c_{ki})$
`ReplaceT()`	$T^a_{ij}T^a_{kl} = \frac{1}{2}\left[\delta_{il}\delta_{jk} - \frac{1}{3}\delta_{ij}\delta_{kl}\right]$
	Special case: $T^a_{ij}T^a_{jl} = C_F\delta_{il}$
`ReplaceD()`	$\delta_{ii} = 3$

Table 18: The replacement rules for the different methods in the class `Color`.

- The data file `Color.dat` will be probed. In case the color factors have been already determined and saved into this file, they will be read in and the `Constructor()` can be terminated.
- If the color factors have not been calculated before, their determination starts with finding out all equal color structures. This is mandatory in order to reduce the number of the color factors to a minimum. It is performed in two steps:
  (a) All color matrices (i.e. `Cfuncs`) will be brought into the same form, i.e. the propagators will be unified. Therefore the numbers of all incoming and outgoing particles will be searched in the list of `Cfuncs` for one diagram. If the appropriate number has been found (every number has to emerge once), a propagator which stands in the same `Cfunc` will be changed to a number beginning from 120 (these numbers do not occur in the usual numbering scheme for propagators, see table 12). All other occurrences of this propagator in the current list of `Cfuncs` will be altered accordingly. This procedure ensures that identical `Cfuncs` in different diagrams have the same numbering order for the propagators.
  (b) Now, the list of `Cfuncs` for all diagrams can be compared. If two list of `Cfuncs` exactly coincide, one of them will be dropped by switching of this diagram. In a list of identities the occurrence of the duplicate color structure will be saved in order to build the full matrix of color factors at the very end.
- Now, all products of the remaining color structures can be evaluated. Loops over the graphs ensure that every diagram will be connected with all other diagrams in order to fill the full matrix of color factors. This is done by multiplying the string form of the color structure for the current diagram with the string of the conjugated color structure of the second diagram. Accordingly, the value of this string has to be determined following these steps:
  (a) Both strings will be translated into a binary string tree with the method `String_Tree::String2Tree()`.
  (b) All structure constants F_{ABC} will be replaced with a combination of generators T^A_{bc} with the method `ReplaceF()`. This is done for both strings separately.

(c) Now, the strings will be multiplied by creating a new `sknot` of the binary string tree, where the operation is a multiplication and the string trees hang on the left- and right-hand side of the new root `sknot`.

(d) All brackets will be expanded (`String_Tree::Expand()`), and the tree is brought into a linear form (`String_Tree::Linear()`). Now, it will be sorted with `String_Tree::Sort()`.

(e) At this stage all generators T^A_{bc} will be replaced or combined to delta functions D_{ab} with `ReplaceT()`.

(f) Now, within the string only constants and delta functions are left. Again, the string tree is expanded, linearized and the delta functions will be replaced with the method `ReplaceD()`.

(g) A string tree, where only constants exist, can be easily evaluated with the method `String_Tree::Color_Evaluate()` and the color factor is now available.

- Having filled the matrix of color factors for the irreducible cases, the full matrix can be determined.
- At the end the matrix of color factors will be saved into a file for later usage. Note, that the user might switch of individual diagrams by means of this matrix, i.e. by setting the corresponding colour factors to zero.

2. `ReplaceF()` replaces all occurrences of the structure constants F_{ABC} with generators T^A_{bc}, see table 18. A loop over the different steps of replacements ensures that at the very end no structure constant is left over. At first the method tries to find combinations of structure constants and generators, i.e. with the same color octet indices and replaces them. This step is performed with the method `Single_ReplaceFT()` which is repeated until no such combination exists any more. Prior to each call of this routine, the string tree is expanded and linearized with the methods `String_Tree::Expand()` and `String_Tree::Linear()`, respectively. Then, if any structure constants are still left, a single structure constant will be replaced with the method `Single_ReplaceF()`. During the replacement new generators appear which can be combined with any remaining structure constants, i.e. the loop starts again.

3. `Single_ReplaceFT()` changes a combination of a structure constant F_{ABC} and a generator T^A_{bc} with one common color octet index into a combination of generators, see table 18. Therefore, an expanded and linearized list is searched for such combinations. This is done recursively in the following way: first of all a multiplication operator is looked for, where a generator or structure constant is attached at one side of the `sknot`. Then, all the other factors of this multiplication series are searched for the appropriate color index. If a match occurs, the two factors will be replaced by the combination of generators. Special care has to be taken for the sign of the product, which depends on the place of the matched color index in the list of arguments for the structure constant. Since the binary tree is searched recursively for such a combination, two calls to `Single_ReplaceFT()` at the end ensure this procedure.

4. `Single_ReplaceF()` replaces a structure constant F_{ABC} by a combination of generators, see table 18. A simple find and replace strategy in the binary tree is the underlying idea of this method. Note that only the first structure constant will be substituted, since in a global strategy it is faster to search for combinations of structure constants and generators (see the proceeding in `ReplaceF()`).

5. `ReplaceT()` substitutes all the combination of two generators with a number of delta functions, see table 18. Since the product of all generators has to yield a complex number, no generator will be left after this method. The procedure is quite simple, since we have an expanded, linearized and sorted binary string tree. Primarily this means that generators with the same color index are immediate neighbours. Therefore, only these two neighbours have to be combined. A further simplification can be achieved, if one of the matrix indices of the two generators are equal. However, in this way all generators will be replaced recursively by calling `ReplaceT()` twice at the end of the method.

6. `ReplaceD()` applies in an expanded and linearized binary string tree all delta functions. This means especially that at the end of this method no delta function should remain (usually this method is called, when no color matrices exist any more). First of all, a delta function will be searched, where two different cases can occur:

   (a) Both arguments of the delta function are equal. Then this delta function will be replaced by a factor of 3 (according to the underlying group structure).

   (b) The arguments could differ. Then, all other delta functions which are multiplied with the current one, are searched for the first argument of the delta function which is then replaced by the second argument. At the end, the current delta function was applied and will be substituted by a factor of 1.

   Two recursive calls to `ReplaceD()` ensure that the whole binary tree is examined.

7. `Mij()` returns the appropriate entry of the color matrix.

`coulomb` calculates the Coulomb factor of an amplitude with two W bosons. This is a first higher-order correction which takes into account a Coulomb force between the two, slow moving, W bosons:

1. The `Constructor()` builds the matrix of Coulomb factors. Note that only products of a graph and a complex conjugate graph will be considered, where both graphs have two W bosons. After the matrix was constructed it will be filled with the method `Build_Matrix()`.

2. `Build_Matrix()` searches for every graph which has two W bosons and marks them with the method `Single_Amplitude::Set_coulomb()`. Then, at each place in the Coulomb matrix, where two marked diagrams meet, the value 1 will be entered.

3. `Calculate()` evaluates the Coulomb factor for a given point in the phase space, i.e. a list of four-momenta for the incoming and outgoing particles with eq. (2.40). For the

calculation the momenta of the two W bosons are mandatory and will be determined with the help of the given four-momenta. Now, all ingredients are at hand for the evaluation of the Coulomb factor.

4. `Factor()` returns one plus the Coulomb factor multiplied with the appropriate matrix element of Coulomb factors.

In the long run, this class will be expanded to top-pairs etc.

4.4 The strings

Within this section all basic methods to construct strings and save them into libraries will be explained. The basic motivation is the following: for every graph and helicity combination one expression consisting of algebraic operations and Z functions exists. Usually the value of this part of the amplitude can be calculated directly, but this has several large disadvantages:

1. Every Z function is calculated in the very moment it is used. Hence some of the Z functions will be calculated more than once.

2. The sequence of operations between the different Z functions has to be determined for every sample of external four-momenta again, since it could not be stored.

3. A lot of helicity combinations can yield zero which is usually reflected in a Z function. Nevertheless, all Z functions belonging to this combination will be calculated because a direct calculation is not very flexible.

These problems can be simply cured by the following steps, respectively:

1. Every Z function will be stored with its value and its list of arguments. Now, every Z function has to be calculated only once at the beginning. This task is performed by the `String_Generator()`. Accordingly, every Z function which was calculated in the class `Elementary_Zfuncs` will now be stored into a list.

2. The whole expression for the calculation can be casted into a character string. Accordingly, for every sample of four-momenta only this string has to be calculated. However, several problems occur during the generation and calculation of this string:

   - The string has to be built during the evaluation of the amplitude which results in a quite complicated program code, since every algebraic operation has to be stored. A simple way out is the usage of a class called `Kabbala` which allows a simultaneous generation of a string during the calculation of complex numbers.
   - The calculation of the string is very time consuming, since this string has to be interpreted, i.e. the algebraic expression has to be examined and evaluated accordingly. The way out is the translation of this string into a binary string tree (`String_Tree`). This tree has an operation saved at each knot and the Z functions at the leafs, i.e. end points of the tree. Now, during the calculation

Class/Struct	Purpose
String_Handler	Handles all string manipulations.
ZXlist	This list will be built from the `String_Generator` and all values necessary for the calculation of a string are stored here.
Virtual_String_Generator	The mother class of all `String_Generators`.
No_String_Generator	Calculating an amplitude without generating any string requires this class.
String_Generator	Translates all Z functions into strings, stores and calculates them later on.
String_Tree	Translates an algebraic equation into a binary tree.
sknot	Is a string knot of this binary tree.
sknotlink	Is a list of `sknots`.
Values	Is the mother class of all connected libraries.
String_Output	Maintains the output into `C++` files.
MyString	A class for handling strings, i.e. adding or searching for strings etc.
Kabbala	Strings (`shem`) and complex numbers (`rishpon`) in one term.

Table 19: A short description of the string classes.

this tree has to be followed only and every operation has to be performed using the values of the left- and the right-hand branch of the appropriate knot. This usually saves a lot of time during the evaluation. However, the by far fastest interpreter of an algebraic expression is the `C++` compiler itself. Accordingly, all strings will be written out into `C++` files, translated and linked for the evaluation. This will be performed within the class `String_Output` and is the optimal way of calculating an expression within `AMEGIC++` .

3. The last problem belongs to the unnecessary calculation of zero parts of the amplitude. Again, the translation into a character string can help to omit these parts. Since all zero Z functions are well known during the first evaluation of the amplitude, all connected parts can be deleted within the character string. Of course, having the string in form of a binary tree relieves this task a lot.

In our description of the single classes we now start with the `String_Handler` which is responsible for the whole organization. Then, we proceed with the `String_Generator` family which takes care for the proper storage of the Z functions. In order to manipulate a string (for instance to delete zero parts), a representation of a string, namely a `String_Tree`, was chosen. This form allows an easy modification and evaluation of the strings. If the strings are at hand, they have to be saved into a library form, i.e. a `C++` file which later can be translated and linked. The classes `Value` and `String_Output` are responsible for this task. Last but not least two helper classes are introduced, one for the handling of strings (`MyString`) and one for tracking algebraic expressions (`Kabbala`). A short description of all these classes can be found in table 19 and their relations are exhibited in figure 22.

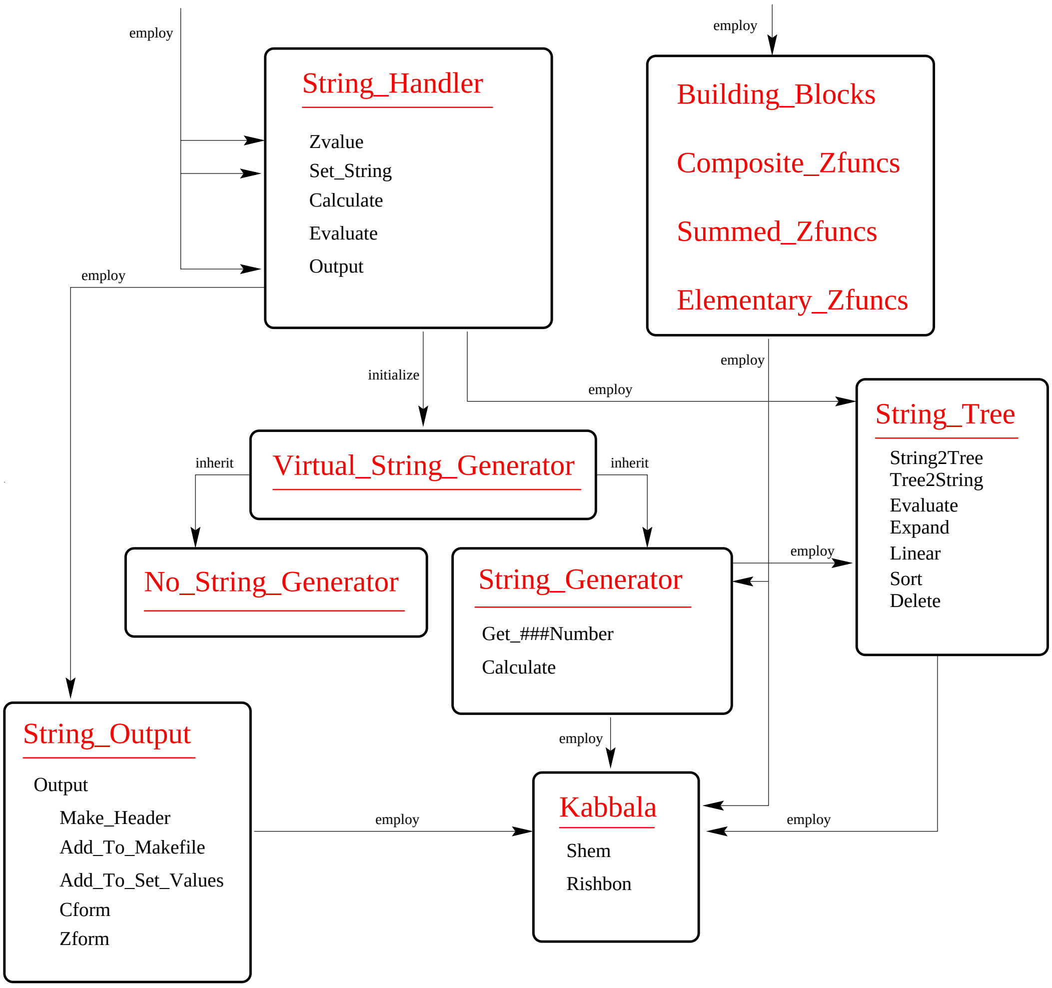


Figure 22: The String classes are responsible for writing the helicity amplitudes into `C++` files.

4.4.1 The handling of strings

The `String_Handler` organizes the whole string generation, the output of the strings into libraries and their calculation. Therefore it not only handles the `String_Generator` for the storage of the Z function but also saves the character strings which have been generated during the evaluation of the amplitudes. These strings can now be translated into a binary tree form. In order to calculate them, a connection between the leafs or end points of the string and the real Z functions has to be drawn which is done as well in this class. At the end an output into `C++` files is performed.

1. The `Constructor` of this class initializes the `String_Generator`, where three different cases could occur:

   (a) No string should be generated at all and therefore a `No_String_Generator` which is a dummy class, will be constructed.

   (b) A string should be generated, i.e. a regular `String_Generator` is constructed, but no library should be used.

(c) An already produced string library shall be used for the evaluation. In this case no string will be generated either. However, if no string library exists, it will be created.

In the latter two cases an identity, i.e. a name, has to be produced in order to label and identify the appropriate library.

2. `Get_Generator()` returns a pointer to the current `String_Generator`.

3. `Init()` initializes a matrix of `sknots`, where every knot represents the root of a `String_Tree`, i.e. a tree representation of a string. The different elements are labeled by the graph number and the appropriate helicity combination. In this way, all character strings for the different parts of the amplitude can be stored.

4. `Set_String()` performs the translation of a given string into a binary string tree, deletes all zero parts and stores the result into the matrix of `sknots`. For this purpose a `String_Tree` object is constructed and filled via the translation method `String_Tree::String2Tree()`. Now, a full binary tree of strings is available, where every knot belongs to an algebraic operation $(+,-,*,/)$ and every end point can be identified with a Z function. However, since all Z functions with the string identity $Z[0]$ yield zero, these parts can now be deleted with `String_Tree::Delete()`. Note that the current `String_Tree()` is a local object only and a translation into a global `String_Tree` object is mandatory which is achieved via `String_Tree::Copy()`. The root of this tree can now be stored into the matrix of `sknots`, where the place is determined through the current graph and helicity number.

5. `Complete()` is performed when all character strings are already stored in the matrix of `sknots`. Here, all Z functions which are attached at the end points of the different binary trees will be connected with the Z functions stored in the `String_Generator`. This step is mandatory, since until now, the list of Z functions was generated independently of the character string representing the algebraic expression for the calculation of one part of the amplitude. In order to calculate the full algebraic equation with the help of the binary tree, the values of these Z functions have to be known. On the other hand, all Z functions are stored in the `String_Generator`. Therefore links between these different representations have to be drawn.

   Within loops over the number of graphs and helicities the binary trees are handled one by one. At first, all leafs are collected with the method `String_Tree::Get_End()`, where a list of `sknots` will be produced. Now, every leaf is connected with the appropriate Z function via `String_Generator::Get_Kabbala()`. At the very end, all Z functions which do not appear in the strings any more (i.e. they belonged to zero parts of the term) are deleted via `Z_Kill()`.

6. `Z_Kill()` simply deletes all Z functions which do not appear in a string any more. Therefore, a loop over all Z functions is performed. Every string from all graphs and helicity combinations is searched for the appropriate function. Of course, since

composed Z functions consist of strings themselves, they have to be searched as well. At the very end, if the Z function could not be found in any of the strings, it will be deleted via `String_Generator::Kill_Z()`.

7. `Calculate()` determines the value of all stored Z functions using the method `String_Generator::Calculate()`. This is the first step of the evaluation of the whole amplitude.

8. `Zvalue()` calculates the value of the string in the form of a binary tree for a given combination of graph and helicity employing the method `String_Tree::Evaluate()`.

9. `Output()` generates the libraries from the strings. Therefore, a `String_Output` object is generated and the output is performed using `String_Output::Output()`.

10. `Set_Values()` chooses the library to be used from a given identity.

11. `Is_String()` returns 1, if all strings are established.

4.4.2 Generating the strings

The structure `ZXlist` contains all informations about a Z function, i.e. all arguments, the value, the type of Z function, a pointer to a `String_Tree` for the composed Z functions and an `on`-switch. Note that we use the label Z-function for a variety of functions which differ by their type, i.e. a real Z function, an X function, a Y function, a complex number, a scalar product of two four-momenta, a S function, a coupling, a propagator, a composed Z function and a mass term. Any algebraic equation for the determination of an amplitude can contain only these possible pieces.

The basic structure for all `String_Generators` is `Virtual_String_Generator` which is a purely virtual class. Therefore it contains only virtual methods which are overwritten in the derived classes. Accordingly, these methods will be described in these classes.

The `No_String_Generator` is a dummy class. Its methods only return the given complex value and are used to keep the same program structure for a pure calculation without the generation of any strings.

The `String_Generator` is used to generate and store all different types of Z functions. The idea is that every Z function depending on its list of arguments is stored only once. For a representation within the algebraic expression in form of a character string these Z functions have to gain a number as well for later recognition during the evaluation. Therefore, the main tasks are comparing a new Z function with all existing ones as well as creating a new one and storing it in the list of `ZXlists` in case it does not yet exist and finally returning an appropriate string representation including the number of the Z function. Accordingly, most methods are concerned with comparing arguments of Z functions, storing and calculating them:

1. The `Constructor()` generates a new list of couplings, flavours and `ZXlists`.

2. `Reset()` deletes the contents of all lists.

3. {Z,X,Y}_Number() has the aim to give a well defined number for any $\{Z, X, Y\}$ function. For this purpose, it compares the list of arguments and couplings for the appropriate function with all already stored $\{Z, X, Y\}$ functions. In case a complete match can be achieved the number of the function is returned, otherwise a -1 will be passed back. In this way a double counting of any function can be omitted.

4. ZX_Count() counts the number of all ZXlists.

5. {Z,X,E}_Count() counts the number of $\{Z, X, E\}$ functions, where the latter one represents a complex number.

6. Get_Cnumber() compares a given coupling with all other couplings in the list. If the coupling could be found, the number within the list is returned, otherwise the new coupling will be attached to the list of couplings.

7. Get_Fnumber() works in the same way as Get_Cnumber(), but for the list of Flavours.

8. Number() constructs a Kabbala (i.e. a string and a complex number in one class) out of a given complex value and a number for the Z function. The string part of the Kabbala is generated in the form of "Z[Z_number]".

9. Get_Znumber() is the method called from the class Elementary_Zfuncs. The input is the list of arguments, the list of couplings and the value of the Z function. First of all, the list of couplings is translated into a list of numbers which hint on a global list via Get_Cnumber(). Then, the number of the Z function in the list of ZXlists is determined with the method Z_Number(). If this Z function already exists, the Kabbala value is returned. Otherwise a new ZXlist will be created. Therefore the lists of arguments and couplings as well as the Kabbala value (via Number) are filled into a ZXlist.

10. Get_Xnumber() works in the same way like Get_Znumber() for X functions.

11. Get_Ynumber() works in the same way like Get_Znumber() for Y functions.

12. Get_Enumber() adds a complex number to the list of ZXlist (this could be the mass of a vector boson for instance). Therefore the existing list is searched for this complex number and a new ZXlist is attached in case it could not be found.

13. Get_Pnumber() adds a propagator term to the list of ZXlists. Here, the complex values and the flavour of the propagator are compared with the existing list.

14. Get_Massnumber() adds a mass term to the list of ZXlists. These terms occur during the transformation of fermion propagators into spinor products. The complex value of the term as well as the flavour are compared with the existing list.

15. Get_Snumber() adds a scalar product of two four-momenta to the list of ZXlists. The two arguments, i.e. the numbers of the four-momenta, are compared with the existing list.

16. `Get_CZnumber()` adds a composed Z function to the list of `ZXlists`. These functions occur, when the building blocks for the calculation of a multiple boson vertex will be stored as an extra Z function. Therefore only the complex values are compared with the already existing list. However, since a composed Z function consists mainly of an algebraic equation in form of a string, this string has to be stored. In order to speed up the evaluation, the string is first of all translated into a binary tree. Then the zero parts are deleted (i.e. where a $Z[0]$ function occurs) and the end points of the tree are connected with the real Z functions. Within the `ZXlist` only the pointer to the root of this tree is stored (a similar treatment can be found within the method `String_Handler::Complete()`).

17. `Calculate()` evaluates all complex values for the different Z functions of the list of `ZXlists`. Accordingly, the different methods of the class `Elementary_Zfuncs` are employed, see table 20. In case a string library is linked this job is performed by `Values::Calculate()`.

18. `Get_Kabbala()` returns a pointer to the `Kabbala` value of a `ZXlist` for a given string representation of a Z function. If the string consists of a complex value only and could not be found in the current list, a new `ZXlist` is created and attached to the list.

19. `Set_EZ()` sets the pointer to the actual `Elementary_Zfunc` object.

20. `ZX_Max_Number()` returns the maximum number of `ZXlists`.

21. `Get_ZXl()` returns a `ZXlist` for a given list number.

22. `Coupl_Max_Number()` returns the maximum number of couplings stored in the list of couplings.

23. `Get_Coupl()` returns the coupling for a given list number.

24. `Kill_Z()` switches a `ZXlist` off.

4.4.3 Translating an equation into a binary string tree

The structure `sknot` represents one algebraic operation and is a knot of the tree for a whole algebraic expression. Therefore it contains a left and a right pointer to the next `sknots`, a string and a pointer to a `Kabbala` value for the end points of the tree and an operation for the knots of the tree.

The structure `sknotlink` is built from a list of `sknots` with a fixed number and contains a link to the next `sknotlink`. The actual number is stored as well.

The `String_Tree` is responsible for the translation of an algebraic expression in form of a character string into a binary tree form. Every knot of the tree contains one operation and the leafs, i.e. end points correspond to the appropriate variables, i.e. for instance Z functions. Consequently, this class not only enables the generation of this tree but also allows its proper manipulation:

1. `Reset()` deletes all linked `sknotlinks` and therefore all `sknots` in the list.

2. `newsk()` provides a new `sknot` from the actual `sknotlink` which saves a fixed number of `sknots`, usually around 200. If all `sknots` are already used up, a new `sknotlink` will be created. In this way a hybrid memory management is achieved, i.e. something between a static and a dynamic memory allocation model.

3. `String2Tree()` translates a given algebraic expression represented by a string into a binary string tree. The string itself will be examined and recursively cut into pieces in the following steps:

   (a) If the string is empty a zero pointer will be returned.

   (b) According to the priorities of the different operators (+,$*$, etc.) the summations and subtractions are examined first. Therefore the string will be searched for global plus and minus signs, where global means that at the very point, where the sign occurs, the number of opening and closing brackets must be equal. If a plus sign is found this way, a new `sknot` will be created with the plus operation. The whole string will be cut into a left- and a right-hand part with respect to the place of the plus sign. Consequently, the left and the right pointer of the `sknot` are set by recursively calling the method `String2Tree()` with the left- and right-hand part of the string. Now, the new `sknot` will be returned. The procedure for the minus sign is the same, only the right-hand side of the string obtains an overall minus sign, and both parts are added.

   (c) Since at this stage all global plus and minus signs are already treated, only global multiplications and divisions can occur. If a global multiplication was found, a new `sknot` is created in the same manner as for the plus sign. The same holds true for global divisions.

   (d) At this stage no global sign should occur any more (expect leading plus and minus signs). Now, an expression can be enclosed by brackets and therefore global brackets which open at the first character of the string and close at the last one are removed and the examination of the strings continues with point (b).

   (e) If no global brackets can be removed, a leading sign could appear, i.e. a plus or minus sign at the first character of the string. In case it is a plus sign it could be simply removed, for a minus sign a new `sknot` will be built with the left side pointing to a `sknot` with a "0" string and the right-hand side to the `sknot` which will be created through a recursive call to `String2Tree()` with the rest of the string as an argument.

   (f) At the very end only a term with no internal structure can remain. This rest string is stored into a new `sknot` which has a zero as operation. Thus it is marked as a leaf of the tree and will be returned.

   In this way the whole string can be translated into a tree and the method returns its root. Note that the different knots of the tree are stored internally in this class.

4. `Color_Evaluate()` calculates the value of a string originating from the determination of the color coefficients of the amplitude. Accordingly, only a distinct number of

different strings could emerge at the end points (i.e. these are integer numbers, color factors like C_F and imaginary units). Therefore they are handled within this method. The evaluation is performed recursively, i.e. the operation of the `sknot` is examined and accomplished accordingly by recursive calls of `Color_Evaluate()` with the knot of the left- and right-hand side, respectively. If the knot is an end point, the string of this `sknot` is compared with the different possibilities and the value is returned.

5. `Evaluate()` calculates the complex value of a string in the same way as `Color_Evaluate()` with a slightly different treatment of the end points. Since this method is used for the evaluation of strings consisting of Z functions, the end points are connected to the real values of the Z functions which are calculated elsewhere (i.e. in `String_Generator`). Therefore the value of every end point is obtained by using the pointer value of the `sknot`.

6. `Get_End()` is used for linking the end points of the tree with the Z functions. Consequently, a list of all end points is mandatory in order to draw the connection. This list is constructed in this method recursively.

7. `Tree2String()` translates a binary tree back into a string. Of course, a tree does not contain any bracket structure and to reestablish these is the main task of this routine. A "granny" knot is used in order to avoid an overburden bracket structure. Now, simple summations do not need an extra bracket, when the granny option was a plus as well. The output is therefore easy to read and often much better than the string which originally was filled into the tree. Note that in its most simplest form the `String_Tree` can be used to simplify the bracket structure of a string term.

8. `Tree2Tex()` works in the same way as `Tree2String()`, with the difference that the output string is LaTeX conformable. This means in particular that the multiplications are replaced by a space and the division is substituted by LaTeX commands.

9. `Delete()` deletes all parts which are connected with a given zero string for a binary string tree. This method is used to drop all zero parts of a helicity combination. Since this can not be cast into a completely recursive procedure, a loop over calls of `Single_Delete()` is performed until all zero parts have been eliminated.

10. `Single_Delete()` deletes recursively all parts of a tree which belong to a zero string:

    (a) If the given `sknot` is zero or an end point the method is terminated with a return.

    (b) If the left-hand knot is an end point and its string equals the given zero string it will be dropped according to the operation of the actual knot:

    - In case it is a multiplication, the actual `sknot` is set equal to the left `sknot`, i.e. the right-hand part is deleted.
    - If the operation is a plus, the actual `sknot` is set to the right-hand side, i.e. the left part is dropped.

    - The most difficult case occurs, when the operation is a minus. Now, the granny knot plays an important role. If no granny exists, i.e. the actual knot is the root, the string of the left knot is set to "0". In case the granny operation is a multiplication, the string of the left knot is set to "0" as well. If it is a plus and the actual knot is on the left-hand side of the granny the string of the left knot is again set to zero, otherwise the granny operation is set to a minus sign and the actual `sknot` is set onto the right-hand side. Last but not least if the granny operation is a minus the same steps as in the plus case will be performed with a slight difference in the changing of the sign of the granny `sknot` to a plus instead of a minus.

   (c) If the left-hand side has not been changed, the right-hand side can be examined. If it is an end point and its string equals the zero string the following manipulations are done:

    - If the operation of the actual `sknot` is a multiplication, the `sknot` is set equal to its right-hand side, i.e. the left part is dropped.
    - In all other cases the actual `sknot` will be set onto the `sknot` on the right-hand side, if the left string is equal "0", otherwise it is set to the `sknot` of the left-hand side.

   At the very end two recursive calls to `Single_Delete()` are performed using the `sknots` of the left- and right-hand side as arguments, respectively.

11. `Expand()` expands an algebraic equation represented by a string tree, i.e. all brackets will be resolved. Since this is not a totally recursive method (as `Delete()`), a loop over calls to `Single_Expand()` is performed until every term is expanded.

12. `Single_Expand()` performs one single expansion, where two different cases can be distinguished:

    - The operator of the mother is a multiplication and the left- or right-hand `sknot` has a plus or minus sign as operation. This is the typical bracket structure in the from of $a * (b \pm c)$. Since a multiplication is a commutative operation the left- and right-hand side will be changed, if the plus or minus sign occurs on the left-hand side. Now, only the right-hand side has to be regarded. At this stage, a special case has to be considered, where inside the brackets only a minus sign emerges $(a * (-b))$. Then, the `sknots` will be rearranged in order to get the form $-a * b$. In the usual case the equation will be simply transformed to $a * b \pm a * c$.
    - The second case is connected with the break-up of extra minus signs, i.e. a piece of equation looks like $a \pm (-b)$. Of course, then the expression will be simplified to gain the form $a \mp b$.

    However, if one of the two cases occur, a switch will be set which ensures that no more manipulations in this step of `Single_Expand()` can be performed. This is necessary, because one manipulation changes the structure of the tree in such a

JHEP02(2002)044

drastic way that the tree has to be reexamined from the very beginning (see `Expand()`). Two recursive calls to `Single_Expand()` with the left- and right-hand side `sknot` as arguments, respectively finalize this method.

13. `Linear()` linearizes the sequence of multiplications, i.e. at the very end all multiplications are lined up. If one draws brackets around every `sknot` with a times operation the following transformation will be done $(a*b)*(c*d) \to a*(b*(c*d))$. This means that in the first case a `sknot` for multiplication with a multiplication on both sides will be transformed to a line on the left-hand side with three subsequent multiplications. This transformation becomes mandatory, if one easily wants to change the sequence of multiplications or wants to examine the different factors in a prescribed way. However, the first criterion is that the actual `sknot` has a times operation. Now, two different cases can appear:

    - The left- and the right-hand side have a multiplication operation, then the transformation is done (by again drawing brackets around every multiplication) in the form of $(a * b) * (c * d) \to c * (d * (a * b))$. Note that the difference to the example above lies in the arbitrariness of the sequence of the times operator.
    - Only the right-hand side has a times operator. Since all multiplications should by construction lined up on the left-hand side, the two sides will be exchanged. In our example this will look like $(b * c) * a \to a * (b * c)$.

    Since this routine is built fully recursive, two calls to `Linear()` with the typical left- and right-hand side `sknot`, respectively, as arguments conclude this method.

14. The `Sort()` method assumes that an already linearized and expanded tree (see `Linear()` and `Expand()`, respectively) exists and sorts the line of multiplications after their strings with help of a standard bubble sort algorithm. This is an interesting option, if one wants to have similar factors side by side. In one step (the method is of course recursive) one line of multiplications will be ordered. An outer loop takes care that the whole line will be perused from the very beginning again and again as long as one exchange within the line was performed. If no more changes occur, the line is considered as sorted. The inner loop compares step by step two subsequent factors, and exchanges them if the first factor is smaller (in a string notation) than the second one. At the very end of the method the classical two recursive calls to `Sort()` will be realized.

15. `Copy()` simply fills a given tree represented by its root `sknot` into the actual tree recursively.

16. `Set_Root()` sets the root of the actual tree.

17. `Get_Root()` returns the root of the actual tree.

As part of further work we envision further simplifications of the strings, like identification of common factors in different amplitudes and their replacement.

4.4.4 Casting strings into C++ files and the string libraries

For every process which is to be linked to AMEGIC++ a separate class (and C++ files) is generated. The class name is deduced from the specific and unique process-id and the class itself is derived from a purely virtual mother class `Values`. This class has two essential methods, namely `Calculate()` and `Evaluate()`. The former one is responsible for the precalculation of the Z functions and the latter one is used to calculate the value for a specified graph and helicity combination. All C++ files as well as the appropriate header files are saved in a directory which is named after the process as well. However, all the different files for the processes will be packed into one library called `Process`. Therefore, the `Makefile` for the production of this library has to be enhanced by the different objects from every process in order to make the linkage as easy as possible. The last step is the inclusion of the calling sequence for this special process. These statements are added to the method `String_Handler::Set_Values()` which is saved into an external file called `Set_Values.C`. Now, the user only has to recompile the new objects via typing `make install`.

The `String_Output` produces the C++ and header files for the output into libraries and manipulates the `Makefile` for their translation and the file `Set_Values.C` for calling the libraries from within the program.

1. The `Constructor()` generates the different identities for the process path and the different files. Especially, all plus and minus signs from the given id will be erased, since these are protected operations within C++.

2. The method `Output()` organizes the whole output to all C++ and header files as well as all manipulations. It is the method which is called from outside (see `String_Handler`):

   - First of all, a header and a C++ filename will be generated with the help of the path and the id strings.
   - With the method `Is_File()` a check is performed, if the header file already exists. In this case the output will be canceled, since it is assumed that the whole library already exists.
   - Now, the header file, strictly speaking the head of the header file, will be generated via `Make_Header()`.
   - With the method `Zform()` the list of Z functions which have to be precalculated at every step, is created, i.e. the method `Values::Calculate()` will be built.
   - The calculation of the different helicity combinations for the different amplitudes is maintained from `Values::Evaluate()`. Furthermore, for every amplitude a new method, named `M#`, where # stands for the graph number, is generated. These C++ files will be written out with the help of the method `Cform()`.
   - Now, all C++ files are established and the header file can be finished. Note that during the output of the C++ files every new method has to be added to the header file for the prescription of the class.

- At the very end the manipulations to the `Set_Values.C` file will be performed via calling `Add_To_Set_Values()`.

3. `Is_File()` returns 1, if the given filename already exists.

4. `Cform()` produces the C++ files for the evaluation of all helicity combinations for all Feynman graphs:

   - First of all, a new C++ file will be generated and its name is added to the list of objects in the `Makefile` with `Add_To_Makefile()`.
   - The main routine which will be called from outside, is `Values::Evaluate()` and will be produced at this stage. From here, the appropriate method for the given graph number will be called. Note that for every graph one method will be generated.
   - Now, a loop over all graphs will be performed in order to produce a new method for each graph. The only argument for these methods is the number of the helicity combination.
   - An equation for each of these will be written out using a loop over all helicity combinations. First of all each equation will be translated from its binary tree form into a single character string. Then, this string is written out, where special care is taken that each line has a maximum number of characters by calling `Line_Form()` (this is mandatory, since these string expressions could be quite long).[6]

   At the end a number of C++ files is generated and recorded in the list of objects in the `Makefile`.

5. `Zform()` produces the method `Values::Calculate()` which is responsible for the pre-calculation of the Z functions. Therefore, the main task is the translation of the list of `ZXlists` from the `String_Generator` into C++ statements. First of all, the name of the C++ file is added to the `Makefile` via `Add_To_Makefile()`. Then, a loop over all `ZXlists` is performed, where the calling sequence for the calculation of the actual Z function depends on its type, see table 20. Note that in the same way as during `Cform()` additional care has to be taken for the maximum number of lines. Therefore, new files will be opened if necessary.

6. `Make_Header()` produces the head of the header which means that the class definition and the standard methods will be enlisted.

[6]Since the resulting C++ file can be rather big, an additional complication arises due to a maximum line number for different compiler and machine types, e.g. we had some problems compiling the extra libraries of AMEGIC++ with a Digital Unix compiler. Therefore, the lines have to be counted and the different methods have to be casted into different C++ files. This is quite simple, but sometimes, already one single method is longer than the allowed maximum line number. Then, this method has to be partitioned as well.

Type	Purpose	Calling Sequence
0	X function	`Elementary_Zfuncs::Xcalc()`
1	Z function	`Elementary_Zfuncs::Zcalc()`
2	Constant	The method tries to reinterpret the constant as a coupling or a $1/M^2$ of a vector boson. These are the only constants which appear during the calculation.
3	scalar product	`Elementary_Zfuncs::Scalc()`
4	Y function	`Elementary_Zfuncs::Ycalc()`
5	Propagator	`Elementary_Zfuncs::Pcalc()`
6	Composed function	A composed function will be written out in its string form using the method `Line_Form()`.
7	Mass term	`Elementary_Zfuncs::Mass_Term_Calc()`

Table 20: Calling sequence for the different Z function types.

7. `Line_Form()` writes a given string into a file. Special care is taken to a maximum number of line characters (usually around 70). If the maximum is reached, a line breaking is performed after the next operation (+,−,∗).

8. `Add_To_Set_Values()` adds to the `Set_Values.C` file the call to the newly generated libraries. Therefore, an `include` for the appropriate header file and a calling sequence within the method `String_Handler::Set_Values()` is subjoined.

9. `Add_To_Makefile()` enters the name of a file into the list of objects for a given Makefile. First of all, the given file will be searched (`Search()`) and in case it already exists, the method terminates. Otherwise, the name of the file will be added at two different places, namely for `libProcess_la_SOURCES` and `libProcess_la_OBJECTS`, since these object files should belong to the library `Process`. If the key word is found, the file name will be included in the next line. Note that for this purpose a temporary file will be used which is copied onto the appropriate `Makefile` at the very end (`Copy()`).

10. `Copy()` makes a copy of one file.

11. `Search()` searches for a given string in a file.

4.4.5 Strings and Kabbala

`MyString` is the implementation of a typical string class extended by some methods compared to the standard template libraries (STL):

1. The usual standard `Constructor()` initializes a zero string. A copy constructor version is available as well.

2. `remove()` removes a number of characters out of the string for a given position and the length of the piece to be deleted.

3. `c_str()` returns a standard `C` character string.

4. `length()` returns the length of the string.

5. `substr(long int)` returns a substring beginning at the given position until the end of the string.

6. `substr(long int,long int)` returns a substring beginning at the given position and ending at the position plus the given length of the substring.

7. `find()` returns the position of a given substring within the string. If no representation could be found, a -1 is returned.

8. `insert()` inserts a substring at the given position.

9. `Convert(double)` converts a double value into a string using `Convert(int)`.

10. `Convert(int)` converts an integer value into a string.

11. The `operators` $=, +=,$ `[]`$, +, ==, != , \ll$ do exactly what one might expect. Therefore a special description will be discarded at this point.

The class `Kabbala` is a representation for a string and a complex number in one class. The name of the class originates from the kabbalistic system which attributes to every character in the hebrew alphabet a certain number and tries to interpret the resulting number of words. In our case this class has the big advantage that one can calculate with `Kabbalas` instead of complex numbers and not only gains the numerical result, but also the string representation of the whole calculation. This is extensively used during the generation of the string libraries. Technically, this class provides the user with all possible operations of a complex number in the same syntax, but is taking care for the correct representation in a string form with all possible preferences of the operators and the appropriate use of brackets. Therefore it can be easily used instead of complex numbers. Two extra methods `Value()` and `String()` return the complex number or the string representation, respectively.

4.5 The phase space

In this section we will describe the classes and methods used to generate the various channels within `AMEGIC++` and to produce suitable sets of phase space points for the integration. The various classes are summarized in table 21, their connections and interrelations are depicted in figure 23.

4.5.1 Organization: the `Phase_Space_Handler`

The class `Phase_Space_Handler` organizes both the generation of integration channels and the integration by steering the corresponding classes `Phase_Space_Generator` and `Phase_Space_Integrator`, respectively. The methods available within the `Phase_Space_Handler` are:

1. The `Constructor` reads in the parameters relevant for the generation of phase space points via `Read_Parameter()`. Then it initializes all possible selectors for phase space points. The only selector implemented in the moment is a jet measure, i.e. the requirement that the final state particles form distinguishable jets.

Class/Struct	Purpose
Phase_Space_Handler	Handles the generation of integrators and the integration over phase space.
Phase_Space_Integrator	Integrates the phase space.
Phase_Space_Generator	Generates the different channels for multi-channel integration.
Channel_Generator	Generates one channel.
Channel_Basics	Basic functions for channels.
Channel_Elements	Contains all structural elements of channels, e.g. propagators.
Selector	Is the mother class of all selectors which perform cuts in the phase space.
Single_Selector	One single selector.
All_Selector	A list of selectors
JetFinder	A selector which cuts via jet-measures.
Channel	Is the mother of all channels.
Single_Channel	Is one single channel.
Multi_Channel	Is a list of channels and manages the optimization.
Rambo	Is one single channel (flat distribution).
Sarge	Is one single channel (antenna distribution).

Table 21: Classes used for the generation of integration channels and the phase space integration.

2. `Read_Parameter()` reads in parameters relevant for the phase space integration. Currently they consist of the jet scheme plus its $y_{\rm cut}$, the relative error, at which the evaluation of cross section terminates and the strategy used for this calculation, i.e. `Rambo` only, or `Rambo + Sarge`, or multi-channel method with generic, process dependent channels in addition to `Rambo + Sarge`.

3. `Create_Integrator()` organizes the number of channels according to the choices given in the parameters. In case, the user choses to use multi-channel methods with process dependent channels, they are constructed amplitude-wise via the method `Construct()` of the class `Phase_Space_Generator`. For this purpose, the list of `Points` of the corresponding `Amplitude` and the process-identifying string are handed over to `Phase_Space_Generator::Construct()` as well as the external flavours and the amplitude-number.

4. `Drop_Channel()` in general is used to eliminate channels generated in `Create_Integrator()` before the actual calculation starts. Channels are dropped, if

   - they yield the same result under permutation of — at least — two particle momenta. The idea is the following: any channel is correlated to one amplitude and produces a set of momenta accordingly. Thus, any two amplitudes differing by just permuting two identical particles give rise to a new channel which basically yields the same result as the original one. To avoid double counting and to keep

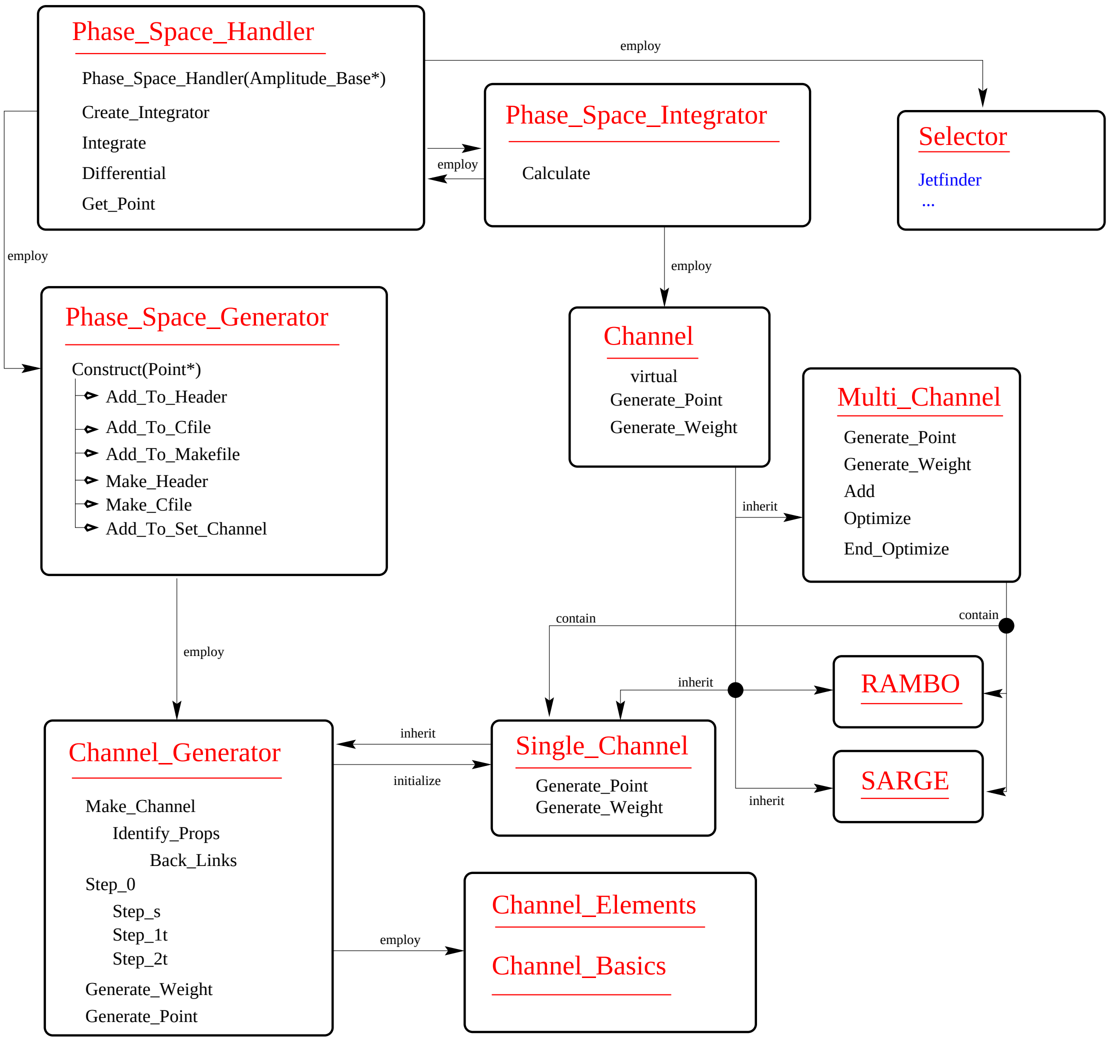


Figure 23: The evaluation and integration of the phase space.

the number of channels small, one of the two channels should be removed. The difference of permutations of momenta and particles is tested via `Compare()`.

- they do not have the maximum number of resonating propagators within the phase space allowed. So, for every channel, the number of potentially resonating propagators is counted, non-maximal number of such propagators lead to the channel being switched off.

5. `Compare()` checks, if two sets of outgoing momenta and particles are just permutations of each other.

6. `Integrate()` calls a suitable method of the `Phase_Space_Integrator` to perform the actual integration. Depending on the number of incoming particles, this is either `Calculate()` or `Calculate_Decay()`.

7. `Differential()` creates one phase space point, i.e. one set of outgoing momenta fulfilling the criteria given by the selectors and returns the corresponding weight.

8. `Get_Point()` returns one phase space point, i.e. one set of four-momenta for the incoming and outgoing particles.

4.5.2 Channels

Since all of the integration within `AMEGIC++` is performed by invoking channels, we start the presentation of the various classes utilized in the construction of the phase space integrator and in the calculation by presenting the virtual class `Channel`. It consists of quite a number of virtual methods, used either to generate a phase space point, i.e. a set of four-momenta for the outgoing particles, and the weight associated with it (`Generate_Point()` and `Generate_Weight()`), or for the actual phase space integration (`Add_Point()`,`Variance()`), or used for the running of the multi-channel method.

1. `Generate_Weight()` is a method to generate the phase space weight of a channel for a given set of momenta.

2. `Generate_Point()` is a variety of methods generating phase space points, i.e. sets of momenta.

3. `Add()` adds a channel to the multi-channel integrator.

4. `Drop_Channel()` drops a channel of the ensemble employed during the multi-channel evaluation of the phase space integral.

5. `Optimize()` optimizes the a priori weights α_i of the multi-channel integrator.

6. `End_Optimize()` performs the last optimization step, selecting the set of a priori weights yielding the smallest overall variance in all optimization steps so far.

7. `Name()` returns the name of the channel.

8. `Set_Name()` sets the name of the channel.

9. `Add_Point()` adds the result for a phase space point to the overall result and increments the total number of points by one.

10. `Variance()` returns the variance accumulated in the channel.

11. `Reset()` resets the result etc. accumulated so far.

12. `Reset_Opt()` resets all internal results, variances etc. used for the optimization procedure of the multi-channel strategy.

13. `Number()` returns the number of channels.

14. `Count_Resonances()` counts and returns the maximal number of potentially resonant propagators within one channel.

Directly derived from it is the class `RAMBO`, generating phase space points according to a flat distribution with help of the following methods in addition to the ones outlined above:

1. The `Constructor` initializes an array of helper functions used for the determination of the phase space weight and calculates them. Furthermore, arrays for outgoing momenta squared, their masses and energies are initialized.

2. `Massive_Point()` rescales massless vectors in order to bring them on their mass shells.

3. `Massive_Weight()` calculates the weight related to the rescaling above.

Similarly, the other default channel of `AMEGIC++`, `SARGE` is derived from `Channel` with the additional methods:

1. `Qcd_Antenna()` actually generates a phase space point according to the QCD antenna pattern, once after some first and last — massless — trial vector has been generated. For this purpose `Basic_Antenna()` is employed.

2. `Basic_Antenna()` generates one antenna, i.e. one four vector emitted by a dipole.

3. `Perm_P()` generates a random permutation of a number of integers with help of the P-algorithm in [23].

4. `Polytope()` produces a uniform random distribution ξ_i inside a polytope with $|\xi_k| < 1$, $|\xi_k - \xi_l| < 1$ along the lines of [24].

Last but not least, the integration by the multi-channel method is realized with help of the — again derived — class `Multi_Channel`. The methods employed here have already been described above. However, some of the algorithms change a little bit when having a multi-channel instead of just a single channel. These are:

1. In `Generate_Point()` one channel i is selected randomly according to the a priori weights α_i to generate the point with its corresponding method.

2. The evaluation of the multi-channel weight w for a phase space point is then performed in `Generate_Weight()`. It is given in terms of the individual weights w_i as $1/w = \sum \alpha_i/w_i$.

3. In `Add_Point()` the accumulative result, the result squared and the weight per channel are increased and the number of generated points is incremented.

4.5.3 Generating channels

The generation of the individual channels is steered by the class `Phase_Space_Generator` and actually performed by appropriate methods of the class `Channel_Generator`. The basic idea is to provide for every occurring amplitude one channel which has the advantage that all different peaking structures of a matrix element will be covered. The generated channels are then saved into `C++` files and managed from a newly created class. This object has a similar name like the one of the process and is derived from the class `Single_Channel`. Consequently, the three different methods `Generate_Point()`, `Generate_Weight()` and `Count_Resonance()` have to decide about the current channel and call an appropriate method with the name `C#_Momenta()`, `C#_Weight()` or `C#_Resonances()` (# is the number of the channel) accordingly. Note that these methods will be created within the class `Channel_Generator`. However, we are going to start with the presentation of the `Phase_Space_Generator`, which is responsible for generating one new channel. It mainly consists of methods to initialize and manipulate files:

1. `Construct()` decides, if a channel belonging to the given amplitude has already been realized or should be created. However, the strategy is the following:

    - The full name of the channel is constructed, including its path. Similarly, names for the header and the `C++` file are generated.
    - The methods `Add_To_Header()` and `Add_To_Cfile()` test first of all, if an appropriate header or steering `C++` file already exists. If this is the case, these files will be searched for the current channel name, if it does not exist, the appropriate calling sequence and representation in the header file will be added.
    - Now, two different cases can emerge. If the channel was already generated, a new `Channel` type object will be created and initialized with the method `Set_Channel()`. Then, the channel is ready to use and will be returned.
    - In the second case, a new channel has to be generated. Accordingly, a new object `Channel_Generator` is produced and the channel is created via `Channel_Generator::Make_Channel()`.
    - Now, the new `C++` files can be attached to the list of objects in the appropriate `Makefile` via `Add_To_Makefile()`.
    - If no header file or steering `C++` file exists, the methods `Make_Header()` and `Make_Cfile()` generate them.
    - At the end, the calling sequence for the newly generated channel will be attached to the file `Set_Channel.C` with the method `Add_To_Set_Channel()`.

2. `Add_To_Header()` adds the three methods `C#_Momenta()`, `C#_Weight()`, and `C#_Resonances()`, where `#` stands for the number of the current channel, to a header file. This manipulation is only performed (by copying the old file to a temporary one, adding lines there and copying back) after it has been checked that the header file exist at all, and that these methods have not been implemented yet.

3. `Add_To_Cfile()` similarly adds the calling sequence for these methods to a `C++` file.

4. `Make_Header()` constructs a new header file for this process, where all channels will be stored. Accordingly, all standard methods will be attached.

5. `Make_Cfile()` generates all steering routines for this process which call the different channels appropriately.

6. `Add_To_Makefile()` adds a newly generated `C++` file to the list of objects in the corresponding `Makefile`.

7. `Add_To_Set_Channel()` creates the calling sequence for the newly generated library in the file `Set_Channel.C`.

8. `Is_File()` checks, whether a specific file already exists.

9. `Search()` checks, whether a specific string is within a file.

10. `Copy()` copies one file to another.

Having set up the file system the actual generation of one channel is performed using methods of the class `Channel_Generator` which is derived from `Single_Channel`. All three methods for handling a channel, i.e. one for generating the four-momenta of the outgoing particles, one for calculating the appropriate weight for a given sample of four-momenta and one for specifying the resonant propagators will be created in this class. Accordingly, in every method `C++` commands and calling sequences are written out into a `C++` file resulting in the appropriate methods. Finally these methods look like the example in appendix A.

1. The `Constructor` initializes the individual channel by setting the number of external particles, incoming and outgoing. It copies the point list and then calls `Identify_Props()` in order to mark all t-channel particles.

2. `Identify_Props()`: starting from the first point (which carries the first incoming particle) the points of the list are connected recursively to their previous ones via `Back_Links()`, until, finally the other point related to an incoming particle with the flag `b` $= -1$ is found. Starting from this endpoint, the backward links are followed and intermediate lines which are considered as t-channel propagators, are counted and marked on the way.

3. `Back_Links()` recursively establishes backward links `prev` between the `left` and `right` offsprings of a point and the point itself. If a final point, i.e. a point having no left and right links has `b` $= -1$, i.e. if it is related to an incoming particle, this point is stored as the end point of the incoming line.

4. `Make_Channel()` maintains the generation of all three methods. Therefore it creates the `C++` file and fills in the method named `C#_Momenta()`, `C#_Weight()` and `C#_Resonances()`. After the first two a call to `Step_0()` with accordingly two different options generates the body of the functions. The last method is simply filled with a list of all resonating propagators.

5. `Step_0()` calls either `Step_s()`, `Step_1t()`, or `Step_2t()`, depending on whether zero, one, or two t-channel propagators have been found. So, to some extent `Step_0()` decides about the basic topology of the channel. Note that two different modes are available for the creation of a method, i.e. generating four-momenta and calculating the appropriate weight. In case, there is no t-channel propagator, s is already set as the square of the sum of incoming momenta.

The scheme to distinguish between various vectors and their squares within the channels, is to label them as `p12` and `s12`, respectively, where `1` and `2` are the numbers of the external particles connected to the vectors. These numbers are determined using `Linked_Masses()`.

6. `Step_s()` creates the complete decay sequence of an s-channel propagator. Consequently, this method is built up recursively and one recursion step consists of the decay of one propagator:

   - If the point under consideration is an outgoing particle, nothing will be done.
   - By calling `Linked_Masses()` for both the `left` and the `right` leg, the indices for their vectors and masses are determined. The names for the momenta are constructed accordingly.
   - With the help of `Generate_Masses()` the invariant masses squared for both the `left` and the `right` leg are determined. Note that for propagators these masses have to be generated.
   - Then the current propagator decays isotropically into the corresponding `left` and `right` vectors, an appropriate calling sequence will be attached to the `C++` file.

   Finally, `Step_s()` is called for both the `left` and the `right` leg.

7. `Step_1t()` is the root for a topology with one t-channel propagator. Similarly to `Step_s()` the masses squared for the two outgoing legs are determined calling `Generate_Masses()`. Then their momenta are fixed by invoking the corresponding `T_Channel_X` building blocks, where `X` is replaced by `Weight` or `Momenta` accordingly. Finally, the two outgoing legs can be treated as s-channel propagators by calling `Step_s()`. Strictly speaking this method is quite similar to `Step_s()` but the isotropic decay replaced by a t-channel decay.

8. `Step_2t()` is the key stone for a topology with two t-channel propagators. The only differences to `Step_1t()` are related to the fact that there are three instead of two outgoing particles, and that there are different kinematical regions for the two t-channel propagators, depending on whether their particles are massless or not. This last fact is reflected in invoking one out of a variety of strategies. At this point it should be noted, however, that at the present stage the only strategy implemented is for two massive t-channel propagators fusing into a massive or massless state and the two other outgoing particles being massless. Obviously, further refinements here are left to further work.

9. `Linked_Masses()` generates a string of numbers for a given `Point` which consists of all particles attached to the same branch. Starting from this `Point`, all `left` and `right` links are followed recursively and end points are added to the corresponding string.

10. `Generate_Masses()` basically produces masses squared for a list of propagators along the following algorithm for each:

- By using Linked_Masses() names for the squared masses of each point are generated. If the point corresponds to an outgoing particle, the mass squared is set directly, otherwise, its minimal value is given by the square root of the sum of squares of the outgoing particles connected to it,

$$s_{\rm min}^{ij\dots} = \sqrt{m_i^2 + m_j^2 + \cdots}. \tag{4.1}$$

- Now a loop over all points not treated so far starts.
  - The most resonating propagator among the points left is selected. The contribution of each single propagator is estimated via $1/(M_f\Gamma_f)^2$, where M_f and Γ_f are the mass and width of the propagating flavour.
  - The maximal $s_{\rm max}^{ij\dots}$ for this propagator is evaluated which is the available s minus all other s of propagators already dealt with and minus all minimal $s_{\rm min}$ of the so far untreated propagators.
  - Now the actual s of this propagator is chosen according to either a Breit-Wigner or a simple pole distribution for a massive or massless particle, respectively.

11. Generate_Weight() and Generate_Point() are dummy methods, i.e. they are only used, when new Channels have been created and not linked properly. Then these methods give an error message.

12. Get_Pointlist() returns the list of Points.

13. Init_T() resets all t-flags in a Point list.

4.5.4 Integration

The phase space integration, i.e. the sampling over Monte Carlo generated sets of four-momenta for the outgoing particles is organized by the Phase_Space_Integrator with help of the following methods

1. Within the Constructor(), the number of optimization steps, the number of phase space points per step as well as a maximum number of phase space points to be generated are set.

2. Calculate() actually performs the calculation of a $2 \to n$ cross section depending on a jet measure. Then the channel responsible for the integration is reset and during a loop points, or better the values related to them are added via Channel::Add_Point(). The necessary set of four-momenta as well as the corresponding value of the Feynman amplitude are obtained via the method Differential() of the Phase_Space_Handler. Note that due to the structure outlined above, the channel used can be a multi-channel or just one channel.

3. Up to some different value for the incoming flux, the method Calculate_Decay() is fairly similar and calculates $1 \to n$ decay widths.

4.5.5 Building blocks for the channels, selectors

The basic building blocks for the construction of specific channels have been listed already in table 4. The corresponding methods are organized in the class `Channel_Elements` (where `X` stands for `Momenta` or `Weight`):

1. `Isotropic2_X()` generates momenta or the weight for an isotropic two-body decay.

2. `Isotropic3_X()` generates momenta or the weight for an isotropic three-body decay.

3. `Anisotropic2_X()` generates momenta or the weight for an anisotropic two-body decay.

4. `Massless_Prop_X()` generates a mass squared or the weight for a massless propagator, i.e. according to a simple pole distribution.

5. `Massive_Prop_X()` generates a mass squared or the weight for a massive propagator, i.e. according to a Breit-Wigner distribution.

6. `T_Channel_X()` generates momenta or the weight for a t-channel propagator.

Within these building blocks, some elementary functions are widely used, they are organized in the class `Channel_Basics`:

1. `rotat()` is used to set up 3×3 matrices for spatial rotations or to rotate a vector with such a matrix already set up.

2. `boost()` is used to define a boost along an arbitrary axis or to transform a vector with such a boost.

3. `sqlam(a,b,c)` returns $\sqrt{((a-b-c)^2 - 4bc)}/a$.

4. `tj()` generates a number s according to a simple pole $1/(a \pm s)^\eta$ with the corresponding arguments passed in the call.

5. `hj()` yields the normalization for `tj()`.

6. `tj1()` specializes `tj()` for the case of a pole in the form of $1/(a-s)^\eta$.

7. `hj1()` yields the weight for `tj1()`.

8. `Pseudo_Angle_Cut()` calculates and returns an angular cut for the t-channel methods `T_Channel_X()` above. So it basically constraints the t in the propagator and avoids potential singularities which might occur for massless particles.

To impose additional cuts on the phase space, the virtual class `Selector` is used. It consists of one method only, `Trigger()` yielding a 0 or a 1 depending, on whether the cut was passed or not. This is then used in the sampling to decide, of whether a specific point should be added to the result, or whether a 0 result should be added, in case the point generated failed to pass the cuts.

Class/Struct	Purpose
Switch	Can be ON or OFF.
Model_Type	Gives the type of the model.
Output	Gives the output level.
Data_Pointer	Is a purely virtual class for all `Data`.
Data	Is a template for reading in data.
Run_Parameter	Steers the reading in of parameters and switches.

Table 22: A short description of the classes connected with the handling of parameters and switches.

Derived from this class is a class `All_Selector` which might be used for a non-trivial combination of constraints.

The only selector actually implemented so far is a cut on jets which is passed if all outgoing particles form different jets according to some jet-measure. This selector is organized in the class `Jet_Finder` and consists — in addition to the trigger — of the methods:

1. `Constructor()` initializes the jet finder, i.e. the clustering scheme, the number of vectors etc.
2. `y_jettest()` determines the minimal jet-measure $y_{\min}$ for a number of four-momenta with `ymin()`.
3. `ymin()` gives the minimal jet-measure for the two four-momenta out of a set, having the smallest y_{ij} (determined by calling `jet()`) according to the scheme selected.
4. `durham()` returns the argument in the Durham-algorithm to be compared with $y_{\rm cut}$ $E^2_{\rm c.m.}$.
5. `jade()` returns the argument in the Jade-algorithm to be compared with $y_{\rm cut}E^2_{\rm c.m.}$.
6. `geneva()` returns the argument in the Geneva-algorithm to be compared with $y_{\rm cut}$ $E^2_{\rm c.m.}$.
7. `jet()` is the wrapper of the three methods above. Depending on the jet-scheme selected, for two vectors the argument of the methods above is returned.
8. `recomb()` recombines two specific vectors out of a set according to the E-scheme.

4.6 Parameters and switches

Within the different files for the steering of the program via parameters and switches, a number of different data types can occur. Not only the usual `C++` types `int`, `double` and `string` can be used during the input, but special types were created. Among them, the classes `Switch` and `Model_Type` are the most important. The first one has the two settings `On` and `Off` and is the standard type for all simple switches. With the second one, the model can be specified, where `pure QCD`, `QCD`, `EW` and `SM` are the possible options for a pure QCD model, a QCD model with the particles e^+, e^- and all their interactions with the

QCD particles added, a pure electroweak sector and the whole Standard Model. All these different types have to be handled in one list of parameters and switches. Therefore, an abstract and purely virtual class has to be constructed, from which all other `Data` types can be derived, i.e. a `Data_Pointer`. Now, an abstract template class, `Data`, is used for all different data types which play the role of a template parameter:

1. `Set_Name()` is used to set the name of the variable.
2. `Get_Name()` returns the name of the variable.
3. `Get_Value()` returns the value of the variable.
4. `Set_Value_Direct()` sets the value of the variable directly.
5. `Set_Value()` converts a given string into the current variable type. This template method is specified for the types `Switch`, `Model_Type` and `string`.

The class `Run_Parameter()` includes the main routines for reading in parameter files. Supplementary to this, the class reads in the parameter file `Run.dat` and provides the program with all parameters and switches necessary for one run. Note that the level of output will be steered as well, one can choose between silent, normal and noisy output.

1. `Init()` defines all parameters and switches (i.e. type and name) which can be read in from the file `Run.dat`. Accordingly, a list of `Data_Pointers` is built. Then, all the data will be filled with the method `Read()`.
2. `Read()` reads in a list of parameters and switches from a given data file. This list includes all possible data, where every variable has its own string name. Now, the data file will be searched line by line for every given string name. If a name could be found, the pertaining value is filled into the list accordingly.
3. `Shorten()` deletes all initial and final spaces in a string.

The rest of the methods are used to set or return the different parameters and switches already read in from the file `Run.dat`:

1. `CMS_E()` returns the center of mass system energy.
2. `Set_CMS_E()` is used, to set the CM-energy.
3. `Model_File()` returns the name of the model file (in most cases `Const.dat`).
4. `Model()` returns the `Model_Type`.
5. `Model_Mass()` returns a `Switch::On`, if the masses should be generated by the model.
6. `Masses()` returns a `Switch::On`, if all masses should be taken into account.
7. `Run_Mass()` returns a `Switch::On`, if the masses should be regarded as running.

Class/Struct	Purpose
Random	Some random number generators.
MyTiming	A timer for measuring run times.
vec3d	A three-vector.
vec4d	A four-vector in Minkowski space.
Matrix	A matrix with arbitrary rank.

Table 23: An overview of the helper classes.

8. `Run_Width()` returns the number of the running width scheme to be used, 0 means no running.

9. `Run_Aqed()` returns a `Switch::On`, if α_{QED} should be regarded as running.

10. `Coulomb()` returns a `Switch::On`, if Coulomb effects should be taken into account.

11. `Mass(Flavour,double)` returns the mass for a given `Flavour` of a particle at a given scale. Accordingly, a possible running of the masses will be taken into account in this method as well.

12. `Mass(Flavour)` uses `Mass(Flavour,double)` to yield the mass of a particle at the CM-energy.

13. `Width(Flavour,double)` works like `Mass (Flavour,double)`, but for the width.

14. `Width(Flavour)` works like `Mass(Flavour)`, but for the width.

15. `Picobarn()` returns the conversion factor between $1/\mathrm{GeV}^2$ and *pb*.

16. `Get_Path()` returns the current path for all input data files.

17. `Set_Path()` sets the current path for all input data files.

18. `Output()` returns the output level.

19. `Set_Output()` sets the output level.

4.7 Helpers

A number of helper classes are always necessary for certain purposes. Primarily, a random number generator is mandatory in every Monte Carlo simulation. A tool to measure the elapsed time for the different methods can be used for an optimization and of course a three- and a four-vector should be available as well. Last but not least a matrix class can be used. A short description of the different classes can be found in table 23.
The class `Random` provides a number of different random number generators. An additional option is the possibility, to store the actual state of the appropriate generator. This ensures that after every event a status can be saved and restored later on (for the examination of a special event for instance). Note that all algorithms for the different random number generators are taken from the Numerical Recipes [25].

1. The `Constructor()` initializes the random number generator with a given seed via `Set_Seed()`.
2. `ran1()` returns a random number after an algorithm of Park and Miller with Bays-Durham shuffle and added safeguards, see [25].
3. `ran2()` returns a random number according to a long period random number generator of L'Ecuyer with Bays-Durham shuffle and added safeguards, see [25].
4. `Ran3()` is a standard random number generator, see [25].
5. `Init_Ran3()` initializes the random number generator.
6. `get()` returns a random number calling `Ran3()`.
7. `getNZ()` returns a random number (with `get()`) excluding zero.
8. `Get_Seed()` returns the actual seed.
9. `Set_Seed()` sets a given seed with the method `Init_Ran3()`.
10. `theta()` returns an angle theta which is uniformly distributed in the cosine of this angle.
11. `WriteOutStatus()` writes out every status register of the random number generator.
12. `ReadInStatus()` reads in every status register of the random number generator.

The class `MyTiming` can be used for internal time measurements. The methods of this class are self-explanatory, i.e. `Start()`, `Stop()` and `PrintTime()`.

The class `vec3d` represents an euclidean three-vector. Typical operators, like $+, -, *$ and the cross product between three-vectors and $*, /$ of three-vectors with scalars are defined accordingly. An operator $\ll$ ensures a proper output of a three-vector, other methods are:

1. The `Constructor()` is available in the form of a standard and a copy constructor as well as one constructor which can be assigned with a four-vector (`vec4d`).
2. `abs()` returns the absolute value (length) of the vector.
3. `sqr()` returns the absolute value squared.
4. `operator[]()` returns the element of the vector for a given place number $(1 \dots 3)$.

The class `vec4d` is the implementation of a four-vector in Minkowski space. All the usual operators, i.e. $+, -, *$ between four-vectors and $*, /$ of four-vectors with scalars are implemented. Additional operators are $\ll$ for the output, $==$ and $!=$ for comparing two four-vectors.

1. Three different `Constructor()` are available for the four-vector, i.e. a standard, a copy and a constructor which can be assigned with the energy and a three-vector.

2. `operator[]()` returns the entry of the four-vector for a given index $(0 \ldots 3)$.

3. `abs2()` returns the absolute value squared of the four-vector.

4. `operator+=()` adds another `vec4d`.

5. `operator-=()` subtracts another `vec4d`.

6. `operator*=()` multiplies the current `vec4d` with a scalar.

The template class `Matrix` represents a square matrix with arbitrary rank, where the rank plays the role of the template parameter. The usual multiplications with a matrix, a scalar or a four-vector (if the rank is equal four) are mandatory. Note that not all possible operations of a matrix have been implemented, since not all operations are necessary for our purposes.

1. Two different types of `Constructor()` exist in this class, i.e. a standard and a copy constructor.

2. `operator=()` is the copy operation.

3. `operator[]()` returns a row of the matrix for a given index.

4. `matrix_out()` makes a structured output of the matrix.

5. `rank()` returns the rank of the matrix.

6. `Num_Recipes_Notation()` translates the internal structure of the matrix into a numerical recipes structure. The main difference lies in the starting number for counting arrays, i.e. zero for `AMEGIC++` and one for the numerical recipes.

7. `Amegic_Notation()` translates the numerical recipes structure back into the internal structure.

8. `Diagonalize()` diagonalizes a matrix, i.e. calculates the eigenvalue and the eigenvectors with `jacobi()`. For this purpose a translation into the notation of the numerical recipes beforehand and a retranslation back into the original notation afterwards are performed.

9. `Diagonalize_Sort()` uses `Diagonalize()` in order to determine the eigenvalues and eigenvectors of a matrix. Afterwards they will be sorted according to their eigenvalues.

10. `jacobi()` diagonalizes a matrix, i.e. calculates the eigenvalues and eigenvectors. This method has been borrowed from the numerical recipes [25].

11. `Dagger()` returns the transposed matrix.

5. Installation guide

5.1 Installation

The installation of `AMEGIC++` is quite simple, since a combination of `automake` and `autoconf` was used to generate the `Makefile`'s of the program. The first script translates a number of `Makefile.am`'s, where a rough description of the object files is included, into a `Makefile.in`. The second step is the generation of a script called `configure` from a basic file `configure.in` which is achieved using `autoconf`. This script is able to translate an abstract `Makefile.in` into a proper `Makefile` by including the actual path configuration of the system. Note that these steps have been already performed by the authors of this program. Accordingly, the last step which has to be done by the user itself, is the translation of the `Makefile.in`'s with the script `configure`.

The steps to install `AMEGIC++` are the following:

1. The program can be downloaded in form of the file `AMEGIC++-1.0.tar.gz`. It has to be unpacked with `gzip` and `tar`.

2. By calling the script `configure` all `Makefile`'s will be generated. Note that they are now adjusted to the appropriate directory structure.

3. With the command `make install` all necessary libraries will be built and the executable will be placed into the directory `Amegic`, where it is ready to use.

4. During the run of the program new `C++` files might be generated by `AMEGIC++`. This is the case for the creation of new integration channels or for saving the different helicity combinations in form of a string. However, if this happens, a new translation with `make install` is all the user has to do. Accordingly, all these files will be packed into a library called `Process` and linked as well.

5.2 Running

The program is executed by the script `Amegic` with the directory of the parameter files as an argument. However, the standard directory is `Testrun`, where the files `Run.dat`, `Const.dat`, `Integration.dat`, `Particle.dat` as well as `Processes.dat` should be available. The different possible options are explained in the appropriate tables 24, 25 and 26, whereas the properties of the Standard model particles can be set according to table 27. Note that the particles which are marked as unstable will be provided with calculated decay width in `AMEGIC++`, the specified width in `Particle.dat` will be ignored.

The last step is to generate a list of processes within the data-file `Processes.dat`. Every process is built in the following way: first the list of incoming particles is specified using their kf-codes and an extra minus for anti-particles. The string "`->`" now parts the list of incoming and outgoing particles, where the latter is created in the same way like the incoming list. A simple example for the process $e^- e^+ \to u\bar{u}$ could be "`11 -11 -> 2 -2`". Note that it is also possible to indicate a list of processes which will be handled one by one. Since now all parameters and switches are determined, the program file could be executed.

Variable	Default	Purpose
CMSENERGY	91.	CM-energy in GeV.
MODELFILE	Const.dat	Model data-file.
MODEL	QCD	Model used: `pure QCD`, `QCD`, `EW` and `SM`.
MODELMASS	Off	Take the masses of the SM-particles from the model data-file.
MASSES	On	Quark masses.
RUNMASS	Off	Running quark masses (LO).
RUNWIDTH	0	Running width scheme.
RUNAQED	Off	Running electroweak coupling α_{QED}.
COULOMB	Off	Coulomb corrections.
OUTPUT	NORMAL	Output level: `SILENT`, `NORMAL` or `NOISY`.

Table 24: The parameters in `Run.dat`.

Variable	Default	Purpose
m_up	.005	Mass of up-quark.
m_down	.01	Mass of down-quark.
m_e-	.000511	Mass of electron.
m_charm	1.3	Mass of charm-quark.
m_strange	.170	Mass of strange-quark.
m_mu	.105658	Mass of muon.
m_top	174.	Mass of top-quark.
m_bottom	4.4	Mass of bottom-quark.
m_tau	1.77705	Mass of tauon.
alphaS(M_Z)	.118	$\alpha_S(M_Z)$
v	246.0	VEV of the SM Higgs-field.
m_H_SM	100.0	SM Higgs mass.
alpha_QED(MZ)	128.	$1/\alpha_{\mathrm{QED}}(M_Z)$
SinTW$\hat{2}$	.23124	$\sin^2(\Theta_{\mathrm{Weinberg}})$
lambda	.0	λ of Wolfenstein's CKM.
A	.0	A of Wolfenstein's CKM.
rho	.0	ρ of Wolfenstein's CKM.
eta	.0	η of Wolfenstein's CKM.

Table 25: The Standard Model parameters in `Const.dat`.

Variable	Default	Purpose
YCUT	0.01	y_{cut} for jet-finders.
ERROR	0.01	Allowed error when calculating matrix-elements.
INTEGRATOR	0	Phase space: Rambo=0, Rambo+Sarge=1, Multichannel+Rambo=2, pure Multichannel=3.
JETFINDER	1	Jet-finder: DURHAM=1,JADE=2,GENEVA=3.

Table 26: The parameters in `Integration.dat`.

However, a typical sample main file in `C++` which is included in the distribution as `Amegic/main.C`, should look like:

JHEP02(2002)044

```
#include "Amegic.H" #include "MyTiming.H" #include
"Run_Parameter.H"

using namespace AMEGIC;

Run_Parameter rpa;

int main(int argc,char* argv[]) {

MyTiming testtimer;
testtimer.Start();
string name("Testrun");
if (argc==2) name = string(argv[1]);

rpa.Init(name);

particle_init(name);

testtimer.PrintTime();
Amegic Test(name);

Test.Run();

testtimer.Stop();
testtimer.PrintTime();

}
```

Note that an extra time measurement is performed as well.

6. Summary

Finally, we would like to compare AMEGIC++ with some other programs addressing the same problem of multiparticle production in high-energy collisions. One of the best known examples is the program CompHep [4]. It provides a complete solution, i.e. the generation of matrix elements at the tree-level for a variety of models and the integration over the phase space of the final state particles, and it is easy to access via a self-explaining user interface. However, it has some limitations. First of all, only processes with maximal four particles in the final state can be treated. This is mainly because CompHep relies on the traditional technique of summing and squaring amplitudes. Furthermore, for the numerical integration the default option is to use VEGAS for the generation of flat phase space. For wildly fluctuating matrix elements to be integrated, i.e. processes with resonating propagators, some input by the user is needed to guarantee convergence of the integration. This input

kf-code	Mass	Width	3*e	Y	SU(3)	2*Spin	On	Stab.	Name
1	.01	.0	−1	−1	1	1	1	1	d_quark
2	.005	.0	2	1	1	1	1	1	u_quark
3	.170	.0	−1	−1	1	1	1	1	s_quark
4	1.3	.0	2	1	1	1	1	1	c_quark
5	4.4	.0	−1	−1	1	1	1	1	b_quark
6	174.0	.0	2	1	1	1	1	1	t_quark
11	.000511	.0	−3	−1	0	1	1	1	e-
12	.0	.0	0	1	0	1	1	1	nu_e
13	.10565	.0	−3	−1	0	1	1	1	mu-
14	.0	.0	0	1	0	1	1	1	nu_mu
15	1.777	.0	−3	−1	0	1	1	1	tau-
16	.0	.0	0	1	0	1	1	1	nu_tau
21	.0	.0	0	0	1	2	1	1	gluon
22	.0	.0	0	0	0	2	1	1	photon
23	80.356	2.07	−3	0	0	2	1	1	W-
24	91.188	2.49	0	0	0	2	1	1	Z
25	150.0	.0	0	0	0	0	1	1	h

Table 27: All particles included in `Particle.dat`.

basically consists of picking one diagram which is believed to yield the dominant contribution to the cross section. For example, for the process $e^+e^- \to 4f$ above the W-pair threshold, the default choice converges badly and yields a large deviation from the result obtained by choosing the non-abelian topology with the three gauge boson vertex as the way to generate the four momenta of the outgoing particles. Another program, that falls into the same class as `CompHep` is `Grace` [6]. Other programs, mainly for the evaluation of matrix elements are `FeynArts/FeynCalc` [5], `MADGRAPH` [7], and `O'Mega/WHIZARD` [8]. In all these programs the results have to be interfaced with a phasespace integration program like `WHIZARD`, as is explicitly done already in `O'Mega/WHIZARD`.

In this paper we presented the newly developed matrix element generator `AMEGIC++` which is capable of calculating Feynman amplitudes as well as cross sections at tree level in the framework of the Standard model for the processes:

1. Electron positron annihilations into jets (up to a number of six jets) and
2. the scattering of QCD particles into up to three jets.

We would like to emphasize that the limits on the number of outgoing particles are related to the treatment of processes with a high number of eventually interacting gluons (in its present form, `AMEGIC++` is constrained to a maximal number of 5 gluons).

Two different kinds of problems occur during the calculation of the Feynman amplitude and their integration to yield a cross section. First of all, since the number of outgoing particles is quite large, the resulting number of Feynman diagrams is enormous (for example $e^+ e^- \to q\bar{q}ggg$ yields 384 diagrams in our notation). Therefore, the usual method of summing and squaring the amplitudes "by hand" is not sufficient enough. A way out provides the helicity amplitude method which allows the decomposition of spinor products into their helicity states. Now, only all other parts of the Feynman diagram have to be translated into spinor products accordingly, i.e. for instance propagators and polarization vectors. Having at hand all helicity amplitudes for a given process, the evaluation of the cross section could start. Here, the next problem comes into play. Since the highly dimensional phase space can contain a lot of differently strong peaks (i.e. resonant propagators or soft and collinear gluons with respect to the quarks), the usual method of a Monte Carlo integration with a uniformly distributed phase space has to be abandoned. Accordingly, re-adjusted phase space integrators have to be produced which take care for the whole peak structure of a specified process. One possibility is the multi-channel integration, where every channel can be assigned to a Feynman diagram of the process under consideration. Now, every channel is built to cover the peak structure of a specific Feynman diagram and therefore the whole peak structure is taken into account.

The underlying ideas of the helicity formalism and the multi-channel integration have been presented throughout this paper in different forms, i.e. from a theoretical, algorithmical and technical point of view. Primarily the detailed description of all classes and methods within `AMEGIC++` should enable a possible user to enhance or simply to use the program. Note that the intention of the program is at this stage an integration of matrix elements only, but an extension to generate full partonic events is straightforward. An implementation of `AMEGIC++` into the event generator `APACIC++` [9] is under way.

Of course, `AMEGIC++` is far from being perfect, therfore a number of extensions are planned for the near future:

- The list of Feynman rules and therefore `Models` will be extended to the Two Higgs Doublet and to the Minimal Supersymmetric Standard Model.

- The program will be enabled to calculate polarized cross sections in order to make predictions for a possible next linear collider with polarized electrons and positrons.

- Initial state radiation will be possible for electron positron and QCD parton scatterings. The former one will include the Yennie-Frautschi-Suura approach and the latter one a structure function scheme which makes it necessary to link parton distribution functions (pdf).

- Further technical refinements are in order, for instance parallelization, further simplifications of the strings, more building blocks for the integration channels, more phase space selectors.

We should note that the program `AMEGIC++` is available upon request from the authors, but a Web page is under construction.

Acknowledgments

F.K. and R.K. would like to thank Mike Bisset, Andreas Schälicke, Steffen Schumann and Jan Winter for extensively testing parts of the program and useful comments on both the program and the manual. We appreciate gratefully the careful reading of the manuscript by James Hetherington, Chris Harris and Peter Richardson and their suggestions for its improvement. F.K. would like to thank the Physics Department of the Technion, Israel, where parts of the work were done, for friendly hospitality. R.K. is grateful for the kind hospitality of the Cavendish Laboratory, where large parts of this work have been finalized. We acknowledge financial support by DESY, DAAD, BMBF and GSI.

A. Sample channel for the phase space integration

This is the C++ file which was produced in order to generate the channel given in figure 15. Three different methods will be created, the first one for the generation of the four-momenta, the second one for the calculation of the appropriate weight and the last one for finding resonant propagators within the channel. The last option is used to drop irrelevant channels.

```
#include "P2_5_e_e___s_cb_tau_nu_tau_P.H" using namespace AMEGIC;

void P2_5_e_e___s_cb_tau_nu_tau_P::C1_Momenta(vec4d* pin,vec4d*
pout,
double* ms_out,double* ran)
{
vec4d  p32456 = pin[0] + pin[1];
double s32456 = p32456.abs2();
double s32_min = Max(pa.ycut_ini()*sqr(rpa.CMS_E()),
sqr(sqrt(ms_out[3]) + sqrt(ms_out[2])));
double s456_min = Max(pa.ycut_ini()*sqr(rpa.CMS_E()),
sqr(sqrt(ms_out[4]) + sqrt(ms_out[5])
+ sqrt(ms_out[6])));
double s32_max = sqr(sqrt(s32456)-sqrt(s456_min));
vec4d  p32;
double s32;
s32 = CE.Massless_Prop_Momenta(1.,s32_min,s32_max,ran[1]);
double s456_max = sqr(sqrt(s32456)-sqrt(s32));
vec4d  p456;
double s456;
s456 = CE.Massless_Prop_Momenta(1.,s456_min,s456_max,ran[2]);
double amct  = 1.;
double alpha = 0.5;
double ctmax = 0.;
double ctmin = 2.;
double tmass = Flavour(kf::code(12)).mass();
CE.T_Channel_Momenta(pin[0],pin[1],p32,p456,s32,s456,tmass,alpha,
ctmax,ctmin,amct,0,ran[3],ran[4]);
double s3 = ms_out[3];
```

```
double s2 = ms_out[2];
CE.Isotropic2_Momenta(p32,s3,s2,pout[3],pout[2],ran[5],ran[6]);
double s45_min = Max(pa.ycut_ini()*sqr(rpa.CMS_E()),
sqr(sqrt(ms_out[4]) + sqrt(ms_out[5])));
double s6 = ms_out[6];
double s45_max = sqr(sqrt(s456)-sqrt(s6));
vec4d  p45;
double s45;
s45 = CE.Massless_Prop_Momenta(1.,s45_min,s45_max,ran[7]);
CE.Isotropic2_Momenta(p456,s45,s6,p45,pout[6],ran[8],ran[9]);
double s4 = ms_out[4];
double s5 = ms_out[5];
CE.Isotropic2_Momenta(p45,s4,s5,pout[4],pout[5],ran[10],ran[11]);
}

double P2_5_e_e___s_cb_tau_nu_tau_P::C1_Weight(vec4d* pin,vec4d*
pout,
double* ms_out)
{
double wt = 1.;
vec4d  p32456 = pin[0] + pin[1];
double s32456 = p32456.abs2();
double s32_min = Max(pa.ycut_ini()*sqr(rpa.CMS_E()),
sqr(sqrt(ms_out[3]) + sqrt(ms_out[2])));
double s456_min = Max(pa.ycut_ini()*sqr(rpa.CMS_E()),
sqr(sqrt(ms_out[4]) + sqrt(ms_out[5])
+ sqrt(ms_out[6])));
double s32_max = sqr(sqrt(s32456)-sqrt(s456_min));
vec4d  p32 = pout[3] + pout[2];
double s32 = p32.abs2();
wt *= CE.Massless_Prop_Weight(1.,s32_min,s32_max,s32);
double s456_max = sqr(sqrt(s32456)-sqrt(s32));
vec4d  p456 = pout[4] + pout[5] + pout[6];
double s456 = p456.abs2();
wt *= CE.Massless_Prop_Weight(1.,s456_min,s456_max,s456);
double amct  = 1.;
double alpha = 0.5;
double ctmax = 0.;
double ctmin = 2.;
double tmass = Flavour(kf::code(12)).mass();
wt *= CE.T_Channel_Weight(pin[0],pin[1],p32,p456,tmass,
alpha,ctmax,ctmin,amct,0);
double s3 = ms_out[3];
```

```
double s2 = ms_out[2];
wt *= CE.Isotropic2_Weight(pout[3],pout[2]);
double s45_min = Max(pa.ycut_ini()*sqr(rpa.CMS_E()),
sqr(sqrt(ms_out[4]) + sqrt(ms_out[5])));
double s6 = ms_out[6];
double s45_max = sqr(sqrt(s456)-sqrt(s6));
vec4d  p45 = pout[4] + pout[5];
double s45 = p45.abs2();
wt *= CE.Massless_Prop_Weight(1.,s45_min,s45_max,s45);
wt *= CE.Isotropic2_Weight(p45,pout[6]);
double s4 = ms_out[4];
double s5 = ms_out[5];
wt *= CE.Isotropic2_Weight(pout[4],pout[5]);
if (!IsZero(wt)) wt = 1./wt/pow(2.*M_PI,5*3.-4.);

return wt;
}

int P2_5_e_e___s_cb_tau_nu_tau_P::C1_Resonances(Flavour*& res_fl)
{
res_fl = new Flavour[3];
res_fl[0] = Flavour(kf::code(23));
res_fl[1] = Flavour(kf::code(23));
res_fl[2] = Flavour(kf::code(23));
return 3;
}
```

Note that only the methods of the class `Channel_Elements` will be used, for details see section 4.5.

B. A sample for a Mathematica interfaced function

This is a sample file generated by importing `Mathematica` output for the calculation of a three gluon vertex:

```
#include "Mathematica_Interface.H"

using namespace AMEGIC;

Kabbala Mathematica_Interface::vGGG() {
return Z(1,0)*(X(2,0)-X(2,1))+Z(2,0)*(X(1,2)-X(1,0))
+Z(2,1)*(X(0,1)-X(0,2));
}
```

C. The loop over loops technique

Sometimes it is necessary to have loops within loops to a depth which is not known before run time. Then, a technique comes into play which allows an arbitrary number of inner loops with different starting and ending values. Therefore three different arrays are required, the first one is filled with all beginning values, the second with all ending values and the last one is used to store the current combination of loop variables. Then a loop is performed until the first variable reaches its endpoint. In every step the last variable will be increased (or decreased accordingly). If it gets larger than its endpoint, the variable is reset to its beginning value and the variable at the previous position is increased (or decreased). Consequently, every time a variable reaches its endpoint, the previous one will be increased (or decreased). This procedure can be repeated until the first variable is counted behind its endpoint and the loop ends.

D. Test run output

In this section we present a test run output for the given example options in section 5 and the process $e^- e^+ \to u\bar{u}g$. Note that we use the GNU C++ compiler under Linux on a Pentium III with 800 MHz.

```
Starting Timer Open File: Testrun/Run.dat Open File:
Testrun/Particle.dat Time: 0 s   (clocks=0)
(User: 0.01 s ,System: 0 s ,Children User: 0 s ,Children System: 0)
Open File: Testrun/Const.dat Open File: Testrun/Const.dat No FFS
Vertex included in this Model No SSV Vertex included in this Model
No VVS Vertex included in this Model No SSS Vertex included in
this Model Number of Vertices: 21 1 process(es) ! No
Value-Library available ! Building Topology... Matching of
topologies... ****File Process/2_3_e-_e+___u_ub_G/Color.dat not
found. Finding diagrams with same color structure...

2 different color structures left ++ + File
Process/2_3_e-_e+___u_ub_G/Color.dat saved. 2 Graphs found Open
File: Testrun/Integration.dat

using RAMBO for phase space integration Starting the calculation.
Lean back and enjoy ... . 1:********************************
2:******************************** Gauge(1): 0.0552225 Gauge(2):
0.0552225
Gauge test: 6.28295e-14%
3:********************************
String test: 0%
5000. LO-3-Jet: 1906.09 pb +- 2.32851%
10000. LO-3-Jet: 1946.15 pb +- 1.69762%
```

```
15000. LO-3-Jet: 1962.73 pb +- 1.39437%
20000. LO-3-Jet: 1986.16 pb +- 1.20933%
25000. LO-3-Jet: 1966.02 pb +- 1.08427%
30000. LO-3-Jet: 1965.21 pb +- 0.988323%
result: 1965.21 Stoping Timer Time: 34.93 s   (clocks=3493)
(User: 33.74 s ,System: 0.12 s ,Children User: 0 s ,Children System: 0)
```

Note that this calculation was performed using internal strings for the helicity amplitudes, if the expressions have been saved into a library, the two last lines would look like follows:

```
Time: 16.22 s   (clocks=1622)
(User: 16.22 s ,System: 0 s ,Children User: 0 s ,Children System: 0)
```

One can see that roughly a factor of 4 can be gained by using the saved libraries. A second example (including all fermion masses) for $e^- e^+ \to u\bar{u}ggg$ will be shown:

```
Starting Timer Open File: Testrun/Particle.dat Time: 0 s
(clocks=0)
(User: 0 s ,System: 0 s ,Children User: 0 s ,Children System: 0)
Alpha_S: 0.118035 No FFS Vertex included in this Model No SSV
Vertex included in this Model No VVS Vertex included in this Model
No SSS Vertex included in this Model ... done with it (61). 1
process(es) ! No Value-Library available ! Building Topology...
Decay Table: Decay Table: Matching of topologies...
**************************************************--> File
Process/2_5_e-_e+___u_ub_G_G_G/Color.dat not found. Finding
diagrams with same color structure... (4->0) (7->0) (11->0)
(5->1) (9->1) (6->2) (10->2) (8->3) (14->12) (16->12) (18->12)
(15->13) (17->13) (22->19) (24->19) (27->19) (23->20) (26->20)
(25->21) (30->28) (32->28) (34->28) (31->29) (33->29) (38->35)
(40->35) (43->35) (39->36) (42->36) (41->37) (45->44) (46->44)
(47->44) 15 different color structures left +++++++++++++++
++++++++++++++ +++++++++++++ ++++++++++++ +++++++++++ ++++++++++
+++++++++ ++++++++ +++++++ ++++++ +++++ ++++ +++ ++ + File
Process/2_5_e-_e+___u_ub_G_G_G/Color.dat saved. 48 Graphs found


using RAMBO for phasespace integration Starting the calculation.
Lean back and enjoy ... .
1:**************************************************-->
2:**************************************************--> Gauge(1):
0.000175546 Gauge(2): 0.000175546
Gauge test: 1.38962e-13%
3:**************************************************-->
String test: 1.5439e-14%
```

```
5000. LO-5-Jet: 6.59965 pb +- 5.96623%
10000. LO-5-Jet: 6.87785 pb +- 4.32809%
15000. LO-5-Jet: 6.53779 pb +- 3.51718%
20000. LO-5-Jet: 6.68752 pb +- 3.02409%
25000. LO-5-Jet: 6.67868 pb +- 2.75748%
30000. LO-5-Jet: 6.70715 pb +- 2.52783%
35000. LO-5-Jet: 6.78655 pb +- 2.37905%
40000. LO-5-Jet: 6.78509 pb +- 2.22563%
45000. LO-5-Jet: 6.7962 pb +- 2.10821%
50000. LO-5-Jet: 6.78024 pb +- 1.98829%
100000. LO-5-Jet: 6.79008 pb +- 1.41157%
150000. LO-5-Jet: 6.9083 pb +- 1.17628%
200000. LO-5-Jet: 6.90344 pb +- 1.01139%
250000. LO-5-Jet: 6.895 pb +- 0.90519%
Stoping Timer Time: 15190.9 s   (clocks=1.51909e+06)
(User: 14839.4 s ,System: 37.98 s ,Children User: 0 s ,Children System: 0)
```

Using the string libraries results in a consumated time of

```
Time: 5732.1 s   (clocks=573210)
(User: 5643.49 s ,System: 7.46 s ,Children User: 0 s ,Children
System: 0)
```

accelerating the calculation by roughly a factor of three. A more detailed comparison between the different phase space integrator tools can be found in a forthcoming publication [26].

References

[1] Recent results for the W boson mass and width: ALEPH collaboration, R. Barate et al., *Measurement of W pair production in e^+e^- collisions at 189 GeV*, *Phys. Lett.* **B 484** (2000) 205 [hep-ex/0005043];
DELPHI collaboration, P. Abreu et al., *Measurement of the mass and width of the W boson in e^+e^- collisions at $\sqrt{s} = 189$ GeV*, *Phys. Lett.* **B 511** (2001) 159 [hep-ex/0104047];
L3 collaboration, M. Acciarri et al., *Measurement of the W pair production cross section and W decay branching fractions in e^+e^- interactions at $\sqrt{s} = 189$ GeV*, *Phys. Lett.* **B 496** (2000) 19 [hep-ex/0008026];
OPAL collaboration, G. Abbiendi et al., *W^+W^- production cross section and w branching fractions in e^+e^- collisions at 189 GeV*, *Phys. Lett.* **B 493** (2000) 249 [hep-ex/0009019];
For the Z boson mass and width see: ALEPH collaboration, R. Barate et al., *Measurement of the Z resonance parameters at LEP*, *Eur. Phys. J.* **C 14** (2000) 1;
OPAL collaboration, G. Abbiendi et al., *Precise determination of the Z resonance parameters at LEP: 'Zedometry'*, *Eur. Phys. J.* **C 19** (2001) 587 [hep-ex/0012018];
LEP collaboration, *Combination procedure for the precise determination of Z boson parameters from results of the LEP experiments*, hep-ex/0101027;

Results for the top-quark can be found in: CDF collaboration, T. Affolder et al., *Measurement of the top quark mass with the collider detector at Fermilab*, *Phys. Rev.* **D 63** (2001) 032003 [hep-ex/0006028];
D0 collaboration, B. Abbott et al., *Direct measurement of the top quark mass at D0*, *Phys. Rev.* **D 58** (1998) 052001 [hep-ex/9801025].

[2] One of the many textbooks on the Standard Model is: J.F. Donoghue, E. Golowich and B.R. Holstein, *Dynamics of the standard model*, Cambridge monographs on particle physics, nuclear physics and cosmology **2**, Cambridge University Press, Cambridge UK 1992,

[3] For LHC see LHC Study Group Collaboration, *The Large Hadron Collider: conceptual design*, CERN-AC-95-05-LHC;
For TESLA see F. Richard, J.R. Schneider, D. Trines and A. Wagner, *Tesla technical design report part I. Executive summary*, hep-ph/0106314;
ECFA/DESY LC Physics Working Group collaboration, J.A. Aguilar-Saavedra et al., *Tesla technical design report part III. Physics at an e^+e^- linear collider*, hep-ph/0106315.

[4] A. Pukhov et al., *Comphep: a package for evaluation of Feynman diagrams and integration over multi-particle phase space. User's manual for version 33*, hep-ph/9908288.

[5] T. Hahn, *Generating feynman diagrams and amplitudes with Feynarts 3*, *Comput. Phys. Commun.* **140** (2001) 418 [hep-ph/0012260];
J. Kublbeck, H. Eck and R. Mertig, *Computeralgebraic generation and calculation of Feynman graphs using FeynArts and FeynCalc*, Prepared for 2nd International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High-energy and Nuclear Physics, La Londe Les Maures, France, 13-18 Jan 1992.

[6] F. Yuasa et al., *Automatic computation of cross sections in hep: status of grace system*, *Prog. Theor. Phys. Suppl.* **138** (2000) 18 [hep-ph/0007053].

[7] T. Stelzer and W.F. Long, *Automatic generation of tree level helicity amplitudes*, *Comput. Phys. Commun.* **81** (1994) 357 [hep-ph/9401258].

[8] T. Ohl, *O'mega and Whizard: Monte Carlo event generator generation for future colliders*, hep-ph/0011287.

[9] R. Kuhn, F. Krauss, B. Ivanyi and G. Soff, *Apacic++ 1.0: a parton cascade in C++*, *Comput. Phys. Commun.* **134** (2001) 223 [hep-ph/0004270].

[10] R. Kleiss and W.J. Stirling, *Spinor techniques for calculating $p\bar{p} \to W^{\pm}/Z - 0+$ jets*, *Nucl. Phys.* **B 262** (1985) 235.

[11] A. Ballestrero, E. Maina and S. Moretti, *Heavy quarks and leptons at e^+e^- colliders*, *Nucl. Phys.* **B 415** (1994) 265 [hep-ph/9212246];
A. Ballestrero and E. Maina, *A new method for helicity calculations*, *Phys. Lett.* **B 350** (1995) 225 [hep-ph/9403244].

[12] R. Kleiss and R. Pittau, *Weight optimization in multichannel Monte Carlo*, *Comput. Phys. Commun.* **83** (1994) 141 [hep-ph/9405257].

[13] F. Caravaglios and M. Moretti, *An algorithm to compute born scattering amplitudes without Feynman graphs*, *Phys. Lett.* **B 358** (1995) 332 [hep-ph/9507237];
An implementation can be found in M. Moretti, T. Ohl and J. Reuter, *O'mega: an optimizing matrix element generator*, hep-ph/0102195.

[14] F.A. Berends, R. Pittau and R. Kleiss, *All electroweak four fermion processes in electron-positron collisions*, *Nucl. Phys.* **B 424** (1994) 308 [hep-ph/9404313].

[15] T. Ohl, *Vegas revisited: adaptive Monte Carlo integration beyond factorization*, *Comput. Phys. Commun.* **120** (1999) 13 [hep-ph/9806432].

[16] C.G. Papadopoulos, *Phegas: a phase space generator for automatic cross-section computation*, *Comput. Phys. Commun.* **137** (2001) 247 [hep-ph/0007335].

[17] R. Kleiss, W.J. Stirling and S.D. Ellis, *A new Monte Carlo treatment of multiparticle phase space at high-energies*, *Comput. Phys. Commun.* **40** (1986) 359;
R. Kleiss and W.J. Stirling, *Massive multiplicities and Monte Carlo*, *Nucl. Phys.* **B 385** (1992) 413.

[18] P.D. Draggiotis, A. van Hameren and R. Kleiss, *Sarge: an algorithm for generating QCD antennas*, *Phys. Lett.* **B 483** (2000) 124 [hep-ph/0004047].

[19] D.Y. Bardin, W. Beenakker and A. Denner, *The coulomb singularity in off-shell w pair production*, *Phys. Lett.* **B 317** (1993) 213.

[20] The two choices provided can be found for instance in D.Y. Bardin, A. Leike, T. Riemann and M. Sachwitz, *Energy dependent width effects in e^+e^- annihilation near the z pole*, *Phys. Lett.* **B 206** (1988) 539;
D.Y. Bardin and T. Riemann, *Off-shell W pair production in e^+e^- annihilation: the CC11 process*, *Nucl. Phys.* **B 462** (1996) 3 [hep-ph/9509341];
E.N. Argyres et al., *Stable calculations for unstable particles: restoring gauge invariance*, *Phys. Lett.* **B 358** (1995) 339 [hep-ph/9507216].

[21] L. Garren, I.G. Knowles, T. Sjostrand and T. Trippe, *Monte Carlo particle numbering scheme: in review of particle physics (RPP 2000)*, *Eur. Phys. J.* **C 15** (2000) 205.

[22] T. Ohl, *Drawing Feynman diagrams with LaTeXand Metafont*, *Comput. Phys. Commun.* **90** (1995) 340 [hep-ph/9505351].

[23] D.E. Knuth, *The art of computer programming, vol. 2*, 2nd edition, Addison-Wesley, 1998.

[24] A. van Hameren and R. Kleiss, *A fast algorithm for generating a uniform distribution inside a high-dimensional polytope*, *Comput. Phys. Commun.* **133** (2000) 1 [physics/0003078].

[25] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical recipes in C*, 2nd edition, Cambridge University Press, Cambridge 1994.

[26] F. Krauss, R. Kuhn, G. Soff, in preparation.