

# *FormCalc* 9   User's Guide

---

Sept. 4, 2015   Thomas Hahn

Abstract: *FormCalc* is a *Mathematica* package which calculates and simplifies tree-level and one-loop Feynman diagrams. It accepts diagrams generated with *FeynArts* 3 and returns the results in a way well suited for further numerical or analytical evaluation.

---

The dreadful legal stuff: *FormCalc* is free software, but is not in the public domain. Instead it is covered by the GNU library general public license. In plain English this means:

- 1) We don't promise that this software works. (But if you find any bugs, please let us know!)
- 2) You can use this software for whatever you want. You don't have to pay us.
- 3) You may not pretend that you wrote this software. If you use it in a program, you must acknowledge somewhere in your publication that you've used our code.

If you're a lawyer, you will rejoice at the exact wording of the license at <http://gnu.org/licenses/lgpl.html>.

*FormCalc* is available from <http://feynarts.de/formcalc>.

*FeynArts* is available from <http://feynarts.de>.

*FORM* is available from <http://nikhef.nl/~form>.

*LoopTools* is available from <http://feynarts.de/looptools>.

If you make this software available to others please provide them with this manual, too. There exists a low-traffic mailing list where updates will be announced. Contact [hahn@feynarts.de](mailto:hahn@feynarts.de) to be added to this list.

If you find any bugs, or want to make suggestions, or just write fan mail, address it to:

Thomas Hahn  
Max-Planck-Institut für Physik  
(Werner-Heisenberg-Institut)  
Föhringer Ring 6  
D-80805 Munich, Germany  
e-mail: [hahn@feynarts.de](mailto:hahn@feynarts.de)

## Contents

<b>1</b>	<b>General Considerations</b>	<b>5</b>
<b>2</b>	<b>Installation</b>	<b>7</b>
<b>3</b>	<b>Generating the Diagrams</b>	<b>7</b>
<b>4</b>	<b>Algebraically Simplifying Diagrams</b>	<b>8</b>
4.1	CalcFeynAmp . . . . .	8
4.2	DeclareProcess . . . . .	14
4.3	Clearing, Combining, Selecting . . . . .	16
4.4	Ingredients of Feynman amplitudes . . . . .	18
4.5	Handling Abbreviations . . . . .	23
4.6	More Abbreviations . . . . .	25
4.7	Resuming Previous Sessions . . . . .	27
4.8	Fermionic Matrix Elements . . . . .	28
4.9	Colour Matrix Elements . . . . .	31
4.10	Putting together the Squared Amplitude . . . . .	33
4.11	Polarization Sums . . . . .	34
4.12	Analytic Unsquared Amplitudes . . . . .	36
4.13	Checking Ultraviolet Finiteness . . . . .	37
4.14	Useful Functions . . . . .	38
<b>5</b>	<b>Tools for the Numerical Evaluation</b>	<b>39</b>
5.1	Generating code . . . . .	41
5.2	Running the Generated Code . . . . .	51
5.2.1	Process definition . . . . .	53
5.2.2	Building up phase space . . . . .	54
5.2.3	Variables . . . . .	55
5.2.4	Cuts . . . . .	56
5.2.5	Convolution . . . . .	57
5.2.6	Integration parameters . . . . .	57
5.2.7	Compiling and running the code . . . . .	58

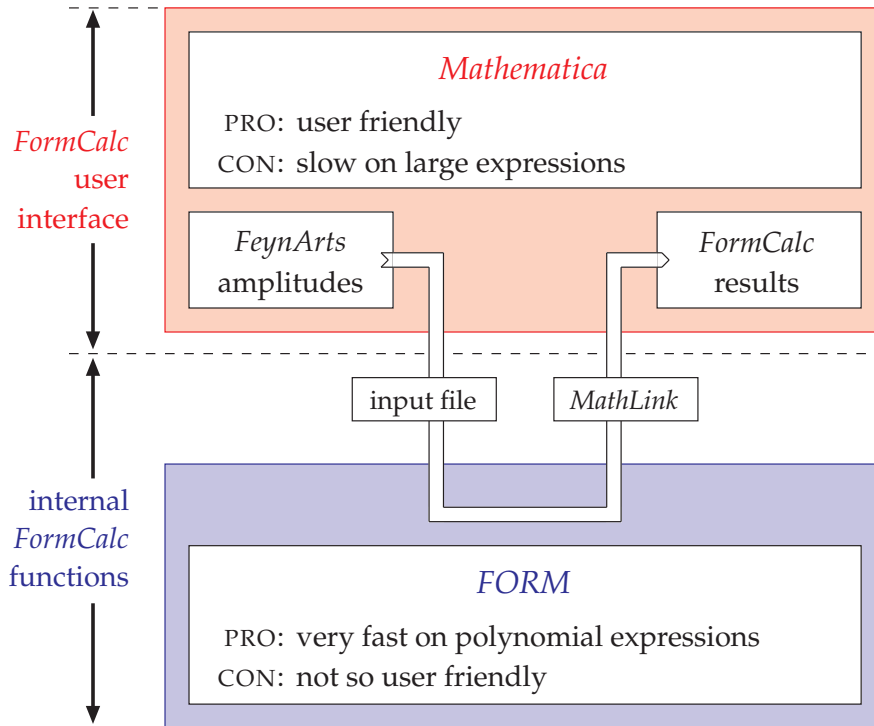
5.2.8	Parallelization and Vectorization . . . . .	60
5.2.9	Scans over parameter space . . . . .	61
5.2.10	Log files, Data files, and Resume . . . . .	63
5.2.11	Shell scripts . . . . .	64
5.3	The Mathematica Interface . . . . .	65
5.3.1	Setting up the Interface . . . . .	66
5.3.2	The Interface Function in Mathematica . . . . .	66
5.3.3	Return values, Storage of Data . . . . .	68
5.3.4	Using the Generated Mathematica Function . . . . .	70
5.4	Renormalization Constants . . . . .	71
5.4.1	Definition of renormalization constants . . . . .	71
5.4.2	Calculation of renormalization constants . . . . .	72
5.5	Infrared Divergences and the Soft-photon Factor . . . . .	74
<b>6</b>	<b>Post-processing of the Results</b>	<b>75</b>
6.1	Reading the data files into <i>Mathematica</i> . . . . .	75
6.2	Special graphics functions for Parameter Scans . . . . .	78
<b>7</b>	<b>Low-level functions for code output</b>	<b>79</b>
7.1	File handling, Type conversion . . . . .	80
7.2	Writing Expressions . . . . .	80
7.3	Variable lists and Abbreviations . . . . .	84
7.4	Declarations . . . . .	87
7.5	Compatibility Functions . . . . .	88

## 1 General Considerations

With the increasing accuracy of experimental data, one-loop calculations have in many cases come to be regarded as the lowest approximation acceptable to publish the results in a respected journal. *FormCalc* goes a big step towards automating these calculations.

*FormCalc* is a *Mathematica* package which calculates and simplifies tree-level and one-loop Feynman diagrams. It accepts diagrams generated with *FeynArts* 3 [Ha00] and returns the results in a way well suited for further numerical (or analytical) evaluation.

Internally, *FormCalc* performs most of the hard work (e.g. working out fermionic traces) in *FORM*, by Jos Vermaseren [Ve00]. A substantial part of the *Mathematica* code indeed acts as a driver that threads the *FeynArts* amplitudes through *FORM* in an appropriate way. The concept is rather straightforward: the symbolic expressions of the diagrams are prepared in an input file for *FORM*, then *FORM* is run, and finally the results are read back into *Mathematica*. The interfacing is completely shielded from the user and is handled internally by the *FormCalc* functions. The following diagram shows schematically how *FormCalc* interacts with *FORM*:

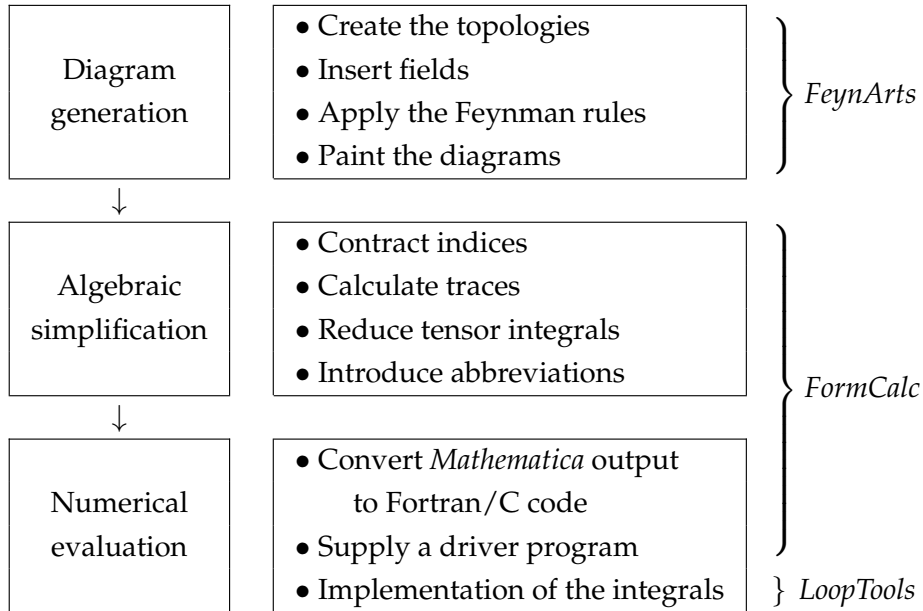


*FormCalc* combines the speed of *FORM* with the powerful instruction set of *Mathematica* and the latter greatly facilitates further processing of the results. Owing to *FORM*'s speed, *FormCalc* can process, for example, the 1000-odd one-loop diagrams of W-W scattering in the Standard Model in a few minutes on ordinary hardware.

One important aspect of *FormCalc* is that it automatically gathers spinor chains, scalar products of vectors, and antisymmetric tensors contracted with vectors, and introduces abbreviations for them. In calculations with non-scalar external particles where such objects are ubiquitous, code produced from the *FormCalc* output (say, in Fortran) can be significantly shorter and faster than without the abbreviations.

*FormCalc* can work in  $D$  and 4 dimensions. In  $D$  dimensions it uses standard dimensional regularization to treat ultraviolet divergences, in 4 dimensions it uses the method of constrained differential renormalization, which at the one-loop level is equivalent to dimensional reduction. Details on these methods can be found in [Ha98].

A one-loop calculation generally includes three steps:



The automation of the calculation is fairly complete in *FormCalc*, i.e. *FormCalc* can eventually produce a complete program to calculate the squared matrix element for a given process. The only thing the user has to supply is a driver program which calls the generated subroutines. The *FormCalc* distribution includes a directory of tools and sample programs which can be modified and/or extended for a particular application. To demonstrate how a full process is calculated, several non-trivial one-loop calculations in the electroweak Standard Model are included in the *FormCalc* package.

It is nevertheless important to realize that the code is generated only at the very end of the calculation (if at all), i.e. the calculation proceeds analytically as far as possible. At all intermediate stages, the results are *Mathematica* expressions which are considerably easier to modify than Fortran or C code.

## 2 Installation

To run *FormCalc* you need *Mathematica* 5 or above, a Fortran compiler, and *gcc*, the GNU C compiler. *FormCalc* comes in a compressed tar archive *FormCalc-n.m.tar.gz*. To install it, create a directory for *FormCalc* and unpack the archive there, e.g.

```
gunzip -c FormCalc-n.m.tar.gz | tar xvf -
cd FormCalc-n.m
./compile
```

The last line compiles the C programs that come with *FormCalc*. The *compile* script puts the binaries in a system-dependent directory, e.g. *Linux*. A single *FormCalc* installation can thus be NFS-mounted on different systems once *compile* has been run on each.

*FormCalc* by default invokes the executable *form* to run *FORM*. If *FORM* is installed under a different name on your system, edit *FormCalc.m* and change the following line:

```
$FormCmd = "form"
```

## 3 Generating the Diagrams

*FormCalc* calculates diagrams generated by *FeynArts* Version 3 or above. Do not use the *FeynArts* function *ToFA1Conventions* to convert the output of *CreateFeynAmp* to the old conventions of *FeynArts* 1.

*FormCalc* can be loaded together with *FeynArts* into one *Mathematica* session. The *FeynArts* results thus need not be saved in a file before calculating them with *FormCalc*.

*FormCalc* can deal with any mixture of fully inserted diagrams and generic diagrams with insertions. The former are produced by *FeynArts* if only one level is requested (via *InsertionLevel*, *AmplitudeLevel*, or *PickLevel*; see the *FeynArts* manual). While both types of input eventually produce the same results, it is important to understand that it is the generic amplitude which is the most costly part to simplify, so using the latter type of diagrams, where many insertions are derived from one generic amplitude, can significantly speed up the calculation.

Usually it is also helpful to organize the diagrams into classes like self-energies, vertex corrections and box corrections; it will also make the amplitudes easier to handle since it reduces their size.

## 4 Algebraically Simplifying Diagrams

### 4.1 CalcFeynAmp

*FeynArts* always produces purely symbolic amplitudes and refrains from simplifying them in any way so as not to be restricted to a certain class of theories. Needless to say, the resulting expressions cannot be used directly e.g. in a Fortran program, but must first be simplified algebraically. The function for this is `CalcFeynAmp`.

`CalcFeynAmp[a1, a2, ...]` calculate the sum of amplitudes  $a_1 + a_2 + \dots$

`CalcFeynAmp` performs the following simplifications:

- indices are contracted as far as possible,
- fermion traces are evaluated,
- open fermion chains are simplified using the Dirac equation,
- colour structures are simplified using the  $SU(N)$  algebra,
- the tensor reduction is performed,
- local terms are added\*,
- the results are partially factored,
- abbreviations are introduced.

The output of `CreateFeynAmp` can be fed directly into `CalcFeynAmp`. Technically, this means that the arguments of `CalcFeynAmp` must be `FeynAmpList` objects. `CalcFeynAmp` is invoked as

```
amps = CreateFeynAmp[...];
result = CalcFeynAmp[amps]
```

The results are returned in the form

`Amp[in -> out] [r1, r2, ...]`

---

\*In  $D$  dimensions, the divergent integrals are expanded in  $\varepsilon = (4 - D)/2$  up to order  $\varepsilon^0$  and the  $\frac{1}{\varepsilon}$  poles are subtracted. The  $\frac{1}{\varepsilon}$  poles give rise to local terms when multiplied with  $D$ 's from outside the integral (e.g. from a  $g^\mu{}_\mu$ ). In 4 dimensions, local terms are added depending on the contractions of indices of the tensor integrals according to the prescription of constrained differential renormalization [dA98].



The lists *in* and *out* in the head of Amp specify the the external particles to which the result belongs. The presence of a particle's mass in the header does not imply that the amplitudes were calculated for on-shell particles.

The actual result is split into parts  $r_1, r_2, \dots$ , such that index sums (marked by SumOver) always apply to the whole of each part. It is possible to extend this splitting also to powers of coupling constants, such that part  $r_i$  has a definite coupling order. To this end one needs to wrap the coupling constants of interest in PowerOf, for example

```
CalcFeynAmp[amp /. g -> g PowerOf[g]]
```

The function PowerOf is there only to keep track of the coupling order and can be replaced by 1 at the end of the calculation.

The full result – the sum of the parts – can trivially be recovered by applying Plus to the outcome of CalcFeynAmp, i.e. Plus@@ CalcFeynAmp[amps].

CalcFeynAmp has the following options:

<i>option</i>	<i>default value</i>	
CalcLevel	Automatic	which level of the amplitude to select (Automatic selects Particles level, if available, otherwise Classes level)
Dimension	D	the space-time dimension in which the calculation is performed (D, 4, or 0)
MomElim	Automatic	how to apply momentum conservation
DotExpand	False	whether to expand terms collected for momentum elimination
NoCostly	False	whether to turn off potentially time-consuming simplifications in FORM
FermionChains	Weyl	how to treat external fermion chains (Weyl, Chiral, or VA)
FermionOrder	Automatic	the preferred ordering of external spinors in Dirac chains
Evanescent	False	whether to keep track of fermionic operators across Fierz transformations
InsertionPolicy	Default	how the level insertions are processed

<i>option</i>	<i>default value</i>	
SortDen	True	whether to sort the denominators of the loop integrals
PaVeReduce	False	whether to reduce tensor to scalar integrals
SimplifyQ2	True	whether to simplify $q^2$ in the numerator
OPP	100	the $N$ in $N$ -point function above which OPP loop integrals are emitted
OPPSlash	False	whether to introduce $\tilde{\mu}$ also on $q$
Gamma5Test	False	whether to substitute $\gamma_5 \rightarrow \gamma_5(1 + \text{Gamma5Test}(D - 4))$
Gamma5ToEps	False	whether to substitute $\gamma_5 \rightarrow \frac{1}{4!}\epsilon_{\mu\nu\rho\sigma}\gamma^\mu\gamma^\nu\gamma^\rho\gamma^\sigma$ in fermion traces
NoExpand	{}	sums containing any of the symbols in the list are not expanded
NoBracket	{}	symbols not to be included in the bracketing in FORM
MomRules	{}	extra rules for transforming momenta
PreFunction	Identity	a function applied to the amplitudes before any simplification
PostFunction	Identity	a function applied to the amplitudes after all simplifications
FileTag	"amp"	the middle part of the temporary FORM file's name
RetainFile	False	whether to retain the temporary FORM input file
EditCode	False	whether to display the FORM code in an editor before sending it to FORM

`CalcLevel` is used to select the desired level in the calculation. In general a diagram can have both `Classes` and `Particles` insertions. The default value `Automatic` selects the deepest level available, i.e. `Particles` level, if available, otherwise `Classes` level.

`Dimension` specifies the space-time dimension in which to perform the calculation. It can take the values `D` and `4`. This is a question of how UV-divergent expressions are treated. The essential points of both methods are outlined in the following. For a more thorough

discussion see [Ha98].

- `Dimension -> D` corresponds to **dimensional regularization** [tH72]. Dimensionally regularizing an expression involves actually two things: analytic continuation of the momenta (and other four-vectors) in the number of dimensions,  $D$ , and an extension to  $D$  dimensions of the Lorentz covariants ( $\gamma_\mu$ ,  $g_{\mu\nu}$ , etc.). The second part is achieved by treating the covariants as formal objects obeying certain algebraic relations. Problems only appear for identities that depend on the 4-dimensional nature of the objects involved. In particular, the extension of  $\gamma_5$  to  $D$  dimensions is problematic. *FormCalc* employs a naive scheme [Ch79] and works with an anticommuting  $\gamma_5$  in all dimensions.
- `Dimension -> 4` selects **constrained differential renormalization** (CDR) [dA98]. This technique cures UV divergences by substituting badly-behaved expressions by derivatives of well-behaved ones in coordinate space. The regularized coordinate-space expressions are then Fourier-transformed back to momentum space. CDR works completely in 4 dimensions. At one-loop level it has been shown [Ha98] to be equivalent to regularization by **dimensional reduction** [Si79], which is a modified version of dimensional regularization: while the integration momenta are still  $D$ -dimensional as in dimensional regularization, all other tensors and spinors are kept 4-dimensional. Although the results are the same, it should be stressed that the conceptual approach in CDR is quite different from dimensional reduction.
- `Dimension -> 0` keeps the whole amplitude  $D$ -dimensional. No rational terms are added and the  $D$ -dependency is expressed through `Dminus4`.

`MomElim` controls in which way momentum conservation is used to eliminate momenta. `False` performs no elimination, an integer between 1 and the number of legs eliminates the specified momentum in favour of the others, and `Automatic` tries all substitutions and chooses the one resulting in the fewest terms.

`DotExpand` determines whether the terms collected for momentum elimination are expanded again. This prevents kinematical invariants from appearing in the abbreviations but typically leads to poorer simplification of the amplitude.

`NoCostly` switches off simplifications in the FORM code which are typically fast but can cause ‘endless’ computations on certain amplitudes.

`FermionChains` determines how fermion chains are returned. `Weyl`, the default, selects Weyl chains. `Chiral` and `VA` select Dirac chains in the chiral ( $P_L/P_R$ ) and vector/axial-vector ( $\mathbb{1}/\gamma_5$ ) decomposition, respectively. Note that to fully evaluate amplitudes containing Dirac chains, helicity matrix elements must be computed with `HelicityME`. For more details on the conceptual treatment of external fermions in *FormCalc*, see [Ha02, Ha04a].

`FermionOrder` specifies an ordering for the external spinors in Dirac chains, i.e. makes sense only in combination with `FermionChains`  $\rightarrow$  Chiral or VA. Possible values are `None`, `Fierz`, `Automatic`, `Colour`, or an explicit ordering, e.g.  $\{2, 1, 4, 3\}$  (corresponding to fermion chains of the form  $\langle 2 | \Gamma | 1 \rangle \langle 4 | \Gamma' | 3 \rangle$ ). `None` applies no reordering. `Fierz` applies the Fierz identities [Ni05] twice, thus simplifying the chains but keeping the original order. `Colour` applies the ordering of the external colour indices (after simplification) to the spinors. `Automatic` chooses a lexicographical ordering (small numbers before large numbers).

The `Evanescence` option toggles whether fermionic operators are tracked across Fierz transformations by emitting terms of the form `Evanescence` [*original operator*, *Fierz operator*].

`InsertionPolicy` specifies how the level insertions are applied and can take the values `Begin`, `Default`, or an integer. `Begin` applies the insertions at the beginning of the FORM code (this ensures that all loop integrals are fully symmetrized). `Default` applies them after simplifying the generic amplitudes (this is fastest). An integer does the same, except that insertions with a `LeafCount` larger than that integer are inserted only after the amplitude comes back from FORM (this is a workaround for the rare cases where the FORM code aborts due to very long insertions).

`SortDen` determines whether the denominators of loop integrals shall be sorted. This is usually done to reduce the number of loop integrals appearing in an amplitude. Sorting may be turned off for testing and in few cases may even lead to shorter amplitudes.

`PaVeReduce` governs the tensor reduction. `False` retains the one-loop tensor-coefficient functions. `Raw` reduces them to scalar integrals but keeps the Gram determinants in the denominator in terms of dot products. `True` simplifies the Gram determinants using invariants.

`SimplifyQ2` controls simplification of terms involving the integration momentum  $q$  squared. If set to `True`, powers of  $q^2$  in the numerator are cancelled by a denominator, except for OPP integrals, where conversely lower- $N$  integrals are put on a common denominator with higher- $N$  integrals to reduce OPP calls, as in:  $N_2/(D_0 D_1) + N_3/(D_0 D_1 D_2) \rightarrow (N_2 D_2 + N_3)/(D_0 D_1 D_2)$ .

`OPP` specifies an integer  $N$  starting from which an  $N$ -point function is treated with OPP methods. For example, `OPP  $\rightarrow$  4` means that  $A$ ,  $B$ ,  $C$  functions are reduced with Passarino–Veltman and  $D$  and up with OPP. A negative  $N$  indicates that the rational terms for the OPP integrals shall be added analytically whereas otherwise their computation is left to the OPP package (CutTools or Samurai).

The integration momentum  $q$  starts life as a  $D$ -dimensional object. In OPP, any  $q^2$  surviving `SimplifyQ2` is substituted by  $q^2 - \tilde{\mu}^2$ , after which  $q$  is considered 4-dimensional. The dimensionful scale  $\tilde{\mu}$  enables the OPP libraries to reconstruct the  $R_2$  rational terms. `OPPQSlash` extends this treatment to the  $\not{q}$  on external fermion chains, i.e. also substitutes  $\not{q} \rightarrow \not{q} + i\gamma_5 \tilde{\mu}$ , where odd powers of  $\tilde{\mu}$  are eventually set to zero.

`Gamma5Test`  $\rightarrow$  `True` substitutes each  $\gamma_5$  by  $\gamma_5(1 + \text{Gamma5Test}(D - 4))$  and it can be tested whether the final result depends on the variable `Gamma5Test` (which it shouldn't).

`Gamma5ToEps`  $\rightarrow$  `True` substitutes all  $\gamma_5$  in fermion traces by  $\frac{1}{4!}\varepsilon_{\mu\nu\rho\sigma}\gamma^\mu\gamma^\nu\gamma^\rho\gamma^\sigma$ . This effectively implements the 't Hooft–Veltman–Breitenlohner–Maison  $\gamma_5$ -scheme. External fermion chains are intentionally exempt since at least the Weyl formalism needs chiral chains. Take care that due to the larger number of Lorentz indices the computation time may increase significantly.

`NoExpand` prohibits the expansion of sums containing certain symbols. In certain cases, expressions can become unnecessarily bloated if all terms are fully expanded, as *FORM* always does. For example, if gauge eigenstates are rotated into mass eigenstates, the couplings typically contain linear combinations of the form  $U_{i1}c_1 + U_{i2}c_2$ . It is not difficult to see that the number of terms generated by the full expansion of such couplings can be considerable, in particular if several of them appear in a diagram. `NoExpand` turns off the automatic expansion, in this example one would select `NoExpand -> U`.

`NoBracket` prevents the given symbols to be included in the internal 'multiplication brackets' in *FORM*. This bracketing is done for performance but prevents the symbols from partaking in further evaluation.

`MomRules` specifies a set of rules for transforming momenta. The notation is that of the final amplitude, i.e.  $k_1, \dots, k_n$  for the momenta,  $e_1, \dots, e_n$  for the polarization vectors.

`PreFunction` and `PostFunction` specify functions to be applied to the amplitude before and after all simplifications have been made. These options are typically used to apply a function to all amplitudes in a calculation, even in indirect calls to `CalcFeynAmp`, such as through `CalcRenConst`.

`RetainFile` and `EditCode` are options used for debugging purposes. The temporary file to which the input for *FORM* is written is not removed when `RetainFile -> True` is set. The name of this file is typically something like `fc-amp-1.frm`. The middle part, `amp`, can be chosen with the `FileTag` option, to disambiguate files from different `CalcFeynAmp` calls. `EditCode` is more comfortable in that it places the temporary file in an editor window before sending it to *FORM*. The command line for the editor is specified in the variable `$Editor`. `EditCode -> Modal` invokes the `$EditorModal` command, which is supposed to be modal (non-detached), i.e. continues only after the editor is closed, thus continuing with possibly modified *FORM* code.

In truly obnoxious cases the function `ReadFormDebug[bits]` can be used to enable debugging output on `stderr` for subsequent `CalcFeynAmp` calls according to the *bits* bit pattern. `ReadFormDebug[bits, file]` writes the output to *file* instead. For currently defined bit patterns and their meaning please see the header of `ReadForm.tm`.

<code>FormPre[amp]</code>	a function to set up the following Form* simplification functions, <i>amp</i> is the raw amplitude
<code>FormSub[subexpr]</code>	a function applied to subexpressions extracted by FORM
<code>FormDot[dotprods]</code>	a function applied to combinations of dot products by FORM
<code>FormMat[matcoeff]</code>	a function applied to the coefficients of matrix elements ( <code>Mat[...]</code> ) in the FORM output
<code>FormNum[numfunc]</code>	a function applied to numerator functions in the FORM output (OPP only)
<code>FormQC[qcoeff]</code>	a function applied to loop-momentum-independent parts of the OPP numerator in the FORM output
<code>FormQF[qfunc]</code>	a function applied to loop-momentum-dependent parts of the OPP numerator in the FORM output
<code>\$FormAbbrDepth</code>	minimum depth an expression has to have to be abbreviated

`CalcFeynAmp` wraps the above functions around various parts of the FORM output for simplification upon return to Mathematica. These are typically relatively short expressions which can be simplified efficiently in Mathematica. The default settings try to balance execution time against simplification efficiency. Occasionally, though, Mathematica will spend excessive time on simplification and in this case one or several of the above should be redefined, e.g. set to `Identity`.

## 4.2 DeclareProcess

For the calculation of an amplitude, many definitions have to be set up internally. This happens in the `DeclareProcess` function.

<code>DeclareProcess[a<sub>1</sub>, a<sub>2</sub>, ...]</code>	set up internal definitions for the calculation of the amplitudes <i>a<sub>1</sub></i> , <i>a<sub>2</sub></i> , ...
--	---

Usually it is not necessary to invoke this function explicitly, as `CalcFeynAmp` does so. All `DeclareProcess` options can be specified with `CalcFeynAmp` and are passed on.

Functions that need the internal definitions set up by `DeclareProcess` are `CalcFeynAmp`, `HelicityME`, and `PolarizationSum`. Invoking `DeclareProcess` directly could be useful e.g. if one needs to change the options between calls to `CalcFeynAmp` and `HelicityME`, or if one wants to call `HelicityME` in a new session without a previous `CalcFeynAmp`.

The output of `DeclareProcess` is a `FormAmp` object.

`FormAmp[proc][amps]`    a preprocessed form of the *FeynArts* amplitudes *amps* for process *proc*

`DeclareProcess` has the following options. They are also accepted by `CalcFeynAmp` in direct invocations (and passed on to `DeclareProcess`) but cannot be set permanently using `SetOptions[CalcFeynAmp, ...]` as they are not `CalcFeynAmp` options.

<i>option</i>	<i>default value</i>	
<code>OnShell</code>	<code>True</code>	whether the external momenta are on shell, i.e. $k_i^2 = m_i^2$
<code>Invariants</code>	<code>True</code>	whether to introduce kinematical invariants like the Mandelstam variables
<code>Transverse</code>	<code>True</code>	whether polarization vectors should be orthogonal to the corresponding momentum, i.e. $\varepsilon_i^\mu k_{i,\mu} = 0$
<code>Normalized</code>	<code>True</code>	whether the polarization vectors should be assumed normalized, i.e. $\varepsilon_i^\mu \varepsilon_{i,\mu}^* = -1$
<code>InvSimplify</code>	<code>True</code>	whether to simplify combinations of invariants as much as possible
<code>Antisymmetrize</code>	<code>True</code>	whether Dirac chains are antisymmetrized

`OnShell` determines whether the external particles are on their mass shell, i.e. it sets  $k_i^2 = m_i^2$  where  $k_i$  is the  $i$ th external momentum and  $m_i$  the corresponding mass. The special value `ExceptDirac` works like `True` except that the Dirac equation is not applied to on-shell momenta, i.e.  $\not{k}_i$  inside fermion chains are not substituted by  $\pm m_i$ .

`Invariants` -> `True` instructs `CalcFeynAmp` to introduce kinematical invariants. In the case of a  $2 \rightarrow 2$  process, these are the familiar Mandelstam variables.

`Transverse` -> `True` enforces orthogonality of the polarization vectors of external vector bosons and their corresponding momentum, i.e. it sets  $\varepsilon_{i,\mu} k_i^\mu = 0$  where  $k_i$  and  $\varepsilon_i$  are the  $i$ th external momentum and polarization vector, respectively.

`Normalized` -> `True` allows `CalcFeynAmp` to exploit the normalization of the polarization vectors, i.e. set  $\varepsilon_i^\mu \varepsilon_{i,\mu}^* = -1$ .

The last four options can concisely be summarized as

Option	Action
OnShell -> True	$k_i \cdot k_i = m_i^2$
Mandelstam -> True	$k_i \cdot k_j = \pm \frac{1}{2} [(s t)_{ij} - m_i^2 - m_j^2]$
Transverse -> True	$\varepsilon_i \cdot k_i = 0$
Normalized -> True	$\varepsilon_i \cdot \varepsilon_i^* = -1$

InvSimplify controls whether CalcFeynAmp should try to simplify combinations of invariants as much as possible.

Antisymmetrize determines whether Dirac chains are antisymmetrized. Note that this does not affect Weyl chains, i.e. has an effect only if FermionChains -> Chiral or VA is chosen in CalcFeynAmp. Take care that antisymmetrized and non-antisymmetrized Dirac chains have an indistinguishable notation. Needless to say, one should not change the antisymmetrization flag between CalcFeynAmp and HelicityME.

### 4.3 Clearing, Combining, Selecting

CalcFeynAmp needs no declarations of the kinematics of the underlying process; it uses the information *FeynArts* hands down. This is convenient, but it also requires certain care on the side of the user because of the abbreviations *FormCalc* automatically introduces in the result (see Sect. 4.5). Owing to the presence of momenta and polarization vectors, abbreviations introduced for different processes will in general have different values, even if they have the same analytical form. To ensure that processes with different kinematics cannot be mixed accidentally, CalcFeynAmp refuses to calculate amplitudes belonging to a process whose kinematics differ from those of the last calculation unless ClearProcess[] is invoked in between.

ClearProcess[]	removes internal definitions before calculating a new process
----------------	---

The Combine function combines amplitudes. It works before and after CalcFeynAmp, i.e. on either FeynAmpList or Amp objects. When trying to combine amplitudes from different processes, Combine issues a warning only, but does not refuse to work as CalcFeynAmp.

Combine[amp <sub>1</sub> , amp <sub>2</sub> , ...]	combines amp <sub>1</sub> , amp <sub>2</sub> , ...
--	--

The following two functions are helpful to select diagrams.



<code>FermionicQ[d]</code>	True if diagram $d$ contains fermions
<code>DiagramType[d]</code>	returns the number of propagators in diagram $d$ not belonging to the loop

`FermionicQ` is used for selecting diagrams that contain fermions, i.e.

```
ferm = CalcFeynAmp[ Select[amps, FermionicQ] ]
```

`DiagramType` returns the number of propagators not containing the integration momentum. To see how this classifies diagrams, imagine a  $2 \rightarrow 2$  process without self-energy insertions on external legs (i.e. in an on-shell renormalization scheme). There, `DiagramType` gives 2 for a self-energy diagram, 1 for a vertex-correction diagram, and 0 for a box diagram, so that for instance

```
vert = CalcFeynAmp[ Select[amps, DiagramType[#] == 1 &] ]
```

calculates all vertex corrections. `DiagramType` is of course only a very crude way of classifying diagrams and not nearly as powerful as the options available in *FeynArts*, like `ExcludeTopologies`.

Individual legs can be taken off-shell with the function `OffShell`.

<code>OffShell[amp, i -&gt; <math>\mu_i</math>, ...]</code>	enforce the relation $p_i^2 = \mu_i^2$ on the amplitudes $amp$ .
---	--

`OffShell[amp, i ->  $\mu_i$ , j ->  $\mu_j$ , ...]` takes legs  $i, j, \dots$  off-shell by substituting the relation  $p_i^2 = \mu_i^2$  for the true on-shell relation  $p_i^2 = m_i^2$ . This is unlike the `CalcFeynAmp` option `OnShell -> False` which takes all legs off-shell by not using  $p_i^2 = m_i^2$  at all.

Finally, the following two functions serve to add factors to particular diagrams.

<code>MultiplyDiagrams[f][amps]</code>	multiply the diagrams in $amps$ with the factor returned by the function $f$
<code>TagDiagrams[amp]</code>	multiply each diagram in $amp$ with the identifier <code>Diagram[n]</code> , where $n$ is the diagram's number

`MultiplyDiagrams[f][amp]` multiplies the diagrams in  $amp$  with factors depending on their contents. The factor is determined by the function  $f$  which is applied to each diagram either as  $f[amplitude]$ , for fully inserted diagrams, or  $f[generic\ amplitude, insertion]$ . For example, to add a QCD enhancement factor to all diagrams containing a quark mass, the following function could be used as `MultiplyDiagrams[QCDfactor][amps]`:

```
QCDfactor[args__] := QCDenh /; !FreeQ[{args}, Mf[3|4, __]]
_QCDfactor = 1
```

`TagDiagrams` is a special case of `MultiplyDiagrams` and multiplies each diagram with an identifier of the form `Diagram[n]`, where  $n$  is the diagram's running number. This provides a very simple mechanism to identify the origin of terms in the final amplitude.

#### 4.4 Ingredients of Feynman amplitudes

The results of `CalcFeynAmp` contain the following symbols:

Momenta and polarization vectors are consecutively numbered, i.e. the incoming momenta are numbered  $k[1] \dots k[n_{\text{in}}]$  and the outgoing momenta  $k[n_{\text{in}} + 1] \dots k[n_{\text{in}} + n_{\text{out}}]$ .

<code>k[n]</code>	$n$ th external momentum
<code>e[n], ec[n]</code>	$n$ th polarization vector and its conjugate
<code>eT[n], eTc[n]</code>	$n$ th polarization tensor and its conjugate
<code>Pair[p, q]</code>	scalar product of the four-vectors $p$ and $q$
<code>Eps[p, q, r, s]</code>	$-i\varepsilon_{\mu\nu\rho\sigma}p^\mu q^\nu r^\rho s^\sigma$ where $\varepsilon$ is the totally antisymmetric Levi-Civita tensor in 4 dimensions with sign convention $\varepsilon^{0123} = +1$
<code>Den[k<sup>2</sup>, m<sup>2</sup>]</code>	the denominator $1/(k^2 - m^2)$
<code>Delta[i, j]</code>	the Kronecker delta $\delta_{ij}$
<code>IGram[d]</code>	the denominator arising from the reduction of a tensor integral, equivalent to $1/d$
<code>Finite</code>	a symbol multiplied with the local terms resulting from $D \cdot (\text{divergent integral})$

Note the extra factor  $-i$  in the definition of `Eps` which is included to reduce the number of explicit  $i$ 's in the final result.

About the use of `Finite`: Whenever a divergent loop integral is multiplied by  $D$  (coming e.g. as  $g_\mu^\mu$  from a self-contracted coupling), local terms arise because the  $\varepsilon$  in  $D = 4 - 2\varepsilon$  cancels the  $1/\varepsilon$ -divergence of the integral, schematically:

$$D \cdot (\text{loop integral}) = 4 \cdot (\text{loop integral}) + (\text{local term}).$$

'Local' refers to the fact that these terms contain no loop integral anymore.

In dimensional regularization, a popular way of checking IR-finiteness of an amplitude is to substitute the loop integrals by their divergent parts and test whether the coefficients of  $\varepsilon^{-1}$  and  $\varepsilon^{-2}$  work out to zero. In *LoopTools*, for example, this is effected by setting `LTLAMBDA` =  $-1$  and  $-2$ , respectively.

The local terms would quite obviously spoil this cancellation and must be set to zero during

the check. To make this possible, CalcFeynAmp multiplies the local terms it generates with the symbol `Finite`, such that `Finite = 1` normally but `Finite = 0` when checking finiteness.

<code>S</code>	Mandelstam variable $s = (k[1] + k[2])^2$
<code>T</code>	Mandelstam variable $t = (k[1] - k[3])^2$
<code>U</code>	Mandelstam variable $u = (k[2] - k[3])^2$
<code>Sij</code>	invariant of the invariant-mass type, $Sij = (k[i] + k[j])^2$
<code>Tij</code>	invariant of the momentum-transfer type, $Tij = (k[i] - k[j])^2$

$SU(N)$  structures are always simplified so that only chains of generators (`SUNT`) remain.

<code>SUNT[a, b, ..., i, j]</code>	the product $(T^a T^b \dots)_{ij}$ of $SU(N)$ -generators, where $a, b, \dots$ are gluon indices ( $1 \dots 8$ ) and $i$ and $j$ are colour indices ( $1 \dots 3$ ); note that <code>SUNT[i, j]</code> represents the identity $\delta_{ij}$ in colour space
<code>SUNT[a, b, ..., 0, 0]</code>	the trace $\text{Tr}(T^a T^b \dots)$

The  $N$  in  $SU(N)$  is specified with the variable `SUNN`.

<i>variable</i>	<i>default value</i>	
<code>SUNN</code>	3	the $N$ in $SU(N)$

Fermionic structures like spinor chains are returned as a `DiracChain` or `WeylChain` object. Lorentz indices connecting two `DiracChains` are denoted by `Lor[n]`, all other Lorentz indices are implicit, e.g. in the contraction with a vector.

DiracChain	the antisymmetrized product of Dirac matrices. The actual Dirac matrix with which an argument is contracted is not written out, e.g. $k[1]$ stands for $\gamma_\mu k[1]^\mu$ inside a DiracChain.
<i>inside a DiracChain:</i>	
1	$\mathbb{1}$
5	$\gamma_5$
6	the chirality projector $P_R = \omega_+ = (\mathbb{1} + \gamma_5)/2$
7	the chirality projector $P_L = \omega_- = (\mathbb{1} - \gamma_5)/2$
$-i$	antisymmetrized chain, $i = 1, 5, 6, 7$ as above
Lor[n]	a Lorentz index connecting two DiracChains
Spinor[p, m, 1]	particle spinor $u(p)$ for which $(\not{p} - m)u(p) = 0$
Spinor[p, m, -1]	antiparticle spinor $v(p)$ for which $(\not{p} + m)v(p) = 0$

It should be especially noted that DiracChains, unlike WeylChains, are antisymmetrized starting from *FormCalc* Version 6. Antisymmetrized chains are typically more convenient for analytical purposes, for example there is the correspondence  $\text{DiracChain}[1, \mu, \nu] = \sigma_{\mu\nu}$ . Note that the antisymmetrization does not extend to the chirality projector or  $\gamma_5$ , e.g.

$$\text{DiracChain}[6, \mu, \nu, \rho] = \frac{\omega_+}{3!} (\gamma_\mu \gamma_\nu \gamma_\rho - \gamma_\mu \gamma_\rho \gamma_\nu - \gamma_\nu \gamma_\mu \gamma_\rho + \gamma_\nu \gamma_\rho \gamma_\mu + \gamma_\rho \gamma_\mu \gamma_\nu - \gamma_\rho \gamma_\nu \gamma_\mu).$$

Weyl chains are quite similar in notation to the Dirac chains. The constituents of a WeylChain always alternate between upper ( $\sigma^{\dot{A}B} = \sigma$ ) and lower indices ( $\sigma_{A\dot{B}} = \bar{\sigma}$ ). The arguments 6 and 7 hence fix the index positions for the entire chain.

WeylChain	the product of sigma matrices. The actual sigma matrix with which an argument is contracted is not written out, e.g. $k[1]$ stands for $\sigma_\mu k[1]^\mu$ inside a WeylChain.
<i>inside a WeylChain:</i>	
6	signifies that the following sigma matrix has upper spinor indices
7	signifies that the following sigma matrix has lower spinor indices
-1	the spinor metric $\epsilon = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$
Spinor[p, m, s]	an undotted 2-spinor
DottedSpinor[p, m, s]	a dotted 2-spinor

*FormCalc* also introduces some model-dependent symbols. It can be argued that this is not a good practice, but the advantage of doing so is just too great to ignore as it can speed up calculations by as much as 15%. This is because e.g.  $MW^2$  is an ordinary symbol while  $MW^{\wedge}2$  is a non-atomic expression (the internal representation is `Power[MW, 2]`).

To change or add to these definitions, edit `FormCalc.m` and find the section beginning with the line

```
(* global definitions for specific models *)
```

Currently, the following model-dependent symbols are introduced by *FormCalc*:

*for the Standard Model:*

Alfa, Alfa2	the fine-structure constant $\alpha = \frac{e^2}{4\pi}$ and its square
Alfas, Alfas2	the “strong fine-structure constant” $\alpha_s = \frac{g_s^2}{4\pi}$ and its square (the spelling with an f was chosen so as not to collide with the CERNlib function ALPHAS2)
CW2, SW2	$\cos^2 \theta_W$ and $\sin^2 \theta_W$
MW2, MZ2, MH2	the squares of various masses
MLE2, ME2, MM2, ML2	
MQU2, MU2, MC2, MT2	
MQD2, MD2, MS2, MB2	
SMSimplify[ <i>expr</i> ]	Simplify with $SW2 \rightarrow 1 - CW2$ and $CW2 \rightarrow MW2/MZ2$

<i>for the MSSM:</i>	
CA2, SA2, CB2, SB2, TB2	$\cos^2 \alpha, \sin^2 \alpha, \cos^2 \beta, \sin^2 \beta,$ and $\tan^2 \beta$
CBA2, SBA2	$\cos^2(\beta - \alpha), \sin^2(\beta - \alpha)$
USfC, UChC, VChC, ZNeuC	the complex conjugates of various mixing matrix elements
MG12, MSf2, MCh2, MNeu2	the squares of various masses
Mh02, MHH2, MA02	
MG02, MHP2, MGp2	
MSSMSimplify[expr]	Simplify with SUSY trigonometric identities (e.g. SBA2 $\rightarrow$ 1 - CBA2)
SUSYTrigExpand[expr]	express various trigonometric symbols (SB2, S2B, etc.) through ca, sa, cb, sb
SUSYTrigReduce[expr]	substitute back ca, sa, cb, sb
SUSYTrigSimplify[expr]	perform trigonometric simplifications by applying SUSYTrigExpand, Simplify, and SUSYTrigReduce in sequence

Often one wants to neglect certain variables, typically masses. Directly defining, say,  $ME = 0$  may lead to problems, however, for instance if  $ME$  appears in a negative power, or in loop integrals where neglecting it may cause singularities. A better way is to assign values to the function `Neglect`, e.g. `Neglect[ME] = 0`, which allows *FormCalc* to replace  $ME$  by zero whenever this is safe. Watch out for the built-in definitions mentioned above: since e.g.  $ME^2$  is automatically replaced by  $ME2$ , one has to assign `Neglect[ME] = Neglect[ME2] = 0` in order to have also the even powers of the electron mass neglected.

<code>Neglect[s] = 0</code>	replace $s$ by 0 except when it appears in negative powers or in loop integrals
-----------------------------	---

To a certain extent it is also possible to use patterns in the argument of `Neglect`. Simple patterns like `_` and `__` work always. Since *FORM*'s pattern matching is far inferior to *Mathematica*'s, though, it is not at all difficult to come up with patterns which are not accepted by *FORM*.

Determining the mass dimension is an easy way of checking consistency of an expression. The function `MassDim` substitutes all symbols in the list `MassDim0` by a random number, all symbols in `MassDim1` by `Mass` times a random number, and all symbols in `MassDim2` by `Mass`<sup>2</sup> times a random number. Symbols not in `MassDim{0,1,2}` are not replaced. The random numbers are supposed to guard against accidental cancellations. An expression consistent in the mass dimension should end up with just one term of the form  $(number) \cdot Mass^n$ .

<code>MassDim[expr]</code>	replace the <code>MassDim<math>n</math></code> -symbols in <code>expr</code> by $(\text{random number}) \cdot \text{Mass}^n$
<code>MassDim0</code>	a list of symbols of mass dimension 0
<code>MassDim1</code>	a list of symbols of mass dimension 1
<code>MassDim2</code>	a list of symbols of mass dimension 2
<code>Mass</code>	a symbol representing the mass dimension

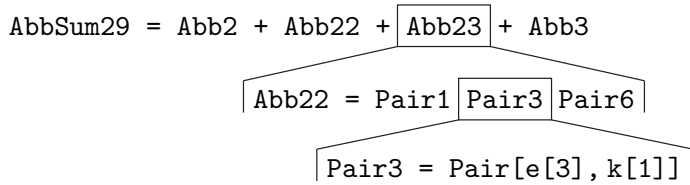
## 4.5 Handling Abbreviations

`CalcFeynAmp` returns expressions where spinor chains, dot products of vectors, and Levi-Civita tensors contracted with vectors have been collected and abbreviated. A term in such an expression may look like

```
COi[cc12, MW2, MW2, S, MW2, MZ2, MW2] *
( -4 AbbSum16 Alfa2 CW2 MW2 S/SW2 + 32 AbbSum28 Alfa2 CW2 S^2/SW2 +
  4 AbbSum30 Alfa2 CW2 S^2/SW2 - 8 AbbSum7 Alfa2 CW2 S^2/SW2 +
  Abb1 Alfa2 CW2 S (T-U)/SW2 + 8 AbbSum29 Alfa2 CW2 S (T-U)/SW2 )
```

The first line stands for the tensor-coefficient function  $C_{12}(M_W^2, M_W^2, s, M_W^2, M_Z^2, M_W^2)$  which is multiplied with a linear combination of abbreviations like `Abb1` or `AbbSum28` with certain coefficients. The coefficients of the abbreviations contain kinematical variables, in this case the Mandelstam variables  $S$ ,  $T$ , and  $U$ , and parameters of the model, here e.g. `Alfa2` or `MW2`. This particular excerpt of code happens to be from a process without external fermions; otherwise spinor chains, abbreviated as `Fn`, would appear, too.

The abbreviations like `Abb1` or `AbbSum29` can drastically reduce the size of an amplitude, particularly so because they are nested in three levels. Consider `AbbSum29` from the example above, which is an abbreviation of about average length:



Without abbreviations, the result would for each `AbbSum29` contain

```
Pair[e[1], e[2]] Pair[e[3], k[1]] Pair[e[4], k[1]] +
Pair[e[1], e[2]] Pair[e[3], k[2]] Pair[e[4], k[1]] +
Pair[e[1], e[2]] Pair[e[3], k[1]] Pair[e[4], k[2]] +
Pair[e[1], e[2]] Pair[e[3], k[2]] Pair[e[4], k[2]]
```

The size-reduction effect can be quantified by comparing the `LeafCount` of the expressions in *Mathematica*. The leaf count is a measure for the size of an expression, more precisely it counts the number of subexpressions or “leaves” on the expression tree. `AbbSum29` has a leaf count of 1 since it is just a plain symbol. In comparison, its fully expanded contents have a leaf count of 77.

<code>Abbr []</code>	the list of abbreviations introduced so far
<code>Abbr [patt]</code>	the list of all abbreviations including (or excluding, if preceded by ! (Not)) the pattern <i>patt</i>
<code>OptimizeAbbr [a]</code>	optimize the abbreviations <i>a</i>
<code>\$OptPrefix</code>	the prefix for additional abbreviations introduced by <code>OptimizeAbbr</code>
<code>SubstSimpleAbbr [expr, abbr]</code>	remove ‘simple’ abbreviations from <i>abbr</i> and substitute them back into <i>expr</i>

The definitions of the abbreviations can be retrieved by `Abbr []` which returns a list of rules such that

```
result //. Abbr[]
```

gives the full, unabbreviated expression. Needless to say, if one wants to use the results of `CalcFeynAmp` outside of the present *FormCalc* session, the abbreviations have to be saved, too, e.g. with

```
Abbr[] >> abbr
```

`Abbr [patt]` retrieves only the subset of abbreviations matching at least one of the patterns *patt*. To exclude, rather than include, a pattern, precede it by ! (Not). For example, `Abbr [a, !b]` returns all abbreviations with *a* but not *b*.

`OptimizeAbbr [a]` optimizes the list of abbreviations *a*. The optimization is done in two steps. First, redundant parts are removed, e.g. the abbreviations

```
AbbSum637 -> Abb109 - Abb187
AbbSum504 -> Abb109 - Abb173 - Abb174 - Abb187
AbbSum566 -> Abb109 + Abb173 + Abb174 - Abb187
```

are replaced by

```
AbbSum637 -> Abb109 - Abb187
AbbSum504 -> AbbSum637 - Abb173 - Abb174
AbbSum566 -> AbbSum637 + Abb173 + Abb174
```



Then, in a second step, common subexpressions are eliminated, thereby simplifying the last lines further to

```
AbbSum637 -> Abb109 - Abb187
Opt1 -> Abb173 + Abb174
AbbSum504 -> AbbSum637 - Opt1
AbbSum566 -> AbbSum637 + Opt1
```

Optimizing the abbreviations may take some time but can also speed up numerical computations considerably. The prefix for the new abbreviations introduced by `OptimizeAbbr`, i.e. the `Opt` in the `OptN` in the example above, can be chosen through the global variable `$OptPrefix`.

`SubstSimpleAbbr[expr, abbr]` removes ‘simple’ abbreviations the list of abbreviations *abbr* and substitutes them back into *expr*. Abbreviations of the form *(number)* and *(number) · (symbol)* are considered simple. Such expressions may show up in the abbreviation list e.g. after further simplification.

## 4.6 More Abbreviations

The abbreviations above are introduced automatically by *FormCalc*, and for specific quantities only. There is also the possibility to abbreviate arbitrary expressions with the `Abbreviate` command.

<code>Abbreviate[expr, lev]</code>	introduce abbreviations for subexpressions of <i>expr</i> starting at level <i>lev</i>
<code>Abbreviate[expr, f]</code>	introduce abbreviations for subexpressions of <i>expr</i> for which <i>f</i> returns True
<code>AbbrevSet[rawexpr]</code>	set up the <code>AbbrevDo</code> function using <i>rawexpr</i> for determining the summation indices
<code>AbbrevDo[expr, i]</code>	introduce abbreviations for subexpressions of <i>expr</i> where <i>i</i> is either a level or a function, as above
<code>\$AbbPrefix</code>	the prefix for abbreviations introduced by <code>Abbreviate</code>

`Abbreviate` introduces abbreviations for subexpressions starting from a given depth of the expression. Depending on this starting level, the expression will be more or less thoroughly abbreviated. A starting level of 1 represents the extreme case where the result of `Abbreviate` is just a single symbol. Currently, only sums are considered for abbreviations.

The alternate invocation, with a function as second argument, introduces abbreviations for

all subexpressions for which this function yields True, like `Select`. This is useful, for example, to get a picture of the structure of an expression with respect to a certain object, as in

```
Abbreviate[a + b + c + (d + e) x, FreeQ[#, x]&, MinLeafCount -> 0]
```

which gives `Sub2 + Sub1 x`, thus indicating that the original expression is linear in `x`.

The functionality of `Abbreviate` is actually separated into a pair of functions `AbbrevSet` and `AbbrevDo`, i.e. `Abbreviate` internally runs `AbbrevSet` to define `AbbrevDo` and then executes `AbbrevDo`. The expression given to `AbbrevSet` is not itself abbreviated but used for determining the summation indices; it could be e.g. a raw amplitude. This is particularly important in cases where partial expressions will be given to `AbbrevDo` and where the summation indices may not be correctly inferred because `AbbrevDo` does not see the full expression. The definition of `AbbrevDo` is not automatically removed, so be careful not to execute `AbbrevDo` with an expression that is not a subexpression of the one given to `AbbrevSet`!

The prefix of the abbreviations are given by the global variable `$AbbPrefix`.

<i>option</i>	<i>default value</i>	
<code>MinLeafCount</code>	10	the minimum leaf count above which a subexpression becomes eligible for abbreviation
<code>Deny</code>	<code>{k, q1}</code>	symbols which must not occur in abbreviations
<code>Fuse</code>	True	whether to fuse adjacent items for which the selection function is True into one abbreviation
<code>Preprocess</code>	Identity	a function applied to subexpressions before abbreviation

`MinLeafCount` determines the minimum leaf count a common subexpression must have in order that a variable is introduced for it.

`Deny` specifies an exclusion list of symbols which must not occur in abbreviations.

`Fuse` specifies whether adjacent items for which the selection function returns True should be fused into one abbreviation. It has no effect when `Abbreviate` is invoked with a depth. For example,

```
Abbreviate[a Pair[1, 2] Pair[3, 4], MatchQ[#, _Pair]&,
  Fuse -> False, MinLeafCount -> 0]
```

introduces two abbreviations, one for `Pair[1, 2]` and one for `Pair[3, 4]`, whereas with

`Fuse` -> True only one abbreviation for the product is introduced.

`Preprocess` specifies a function to be applied to all subexpressions before introducing abbreviations for them.

The abbreviations introduced with `Abbreviate` are returned by `Subexpr []`. This works similar to `Abbr []` and also the `OptimizeAbbr` function can be applied in the same way.

<code>Subexpr []</code>	the list of abbreviations introduced for subexpressions so far
<code>Subexpr [args]</code>	executes <code>Abbreviate [args]</code> locally, i.e. without registering the subexpressions permanently, and returns {list of subexpressions, abbreviated expression}

Introducing abbreviations for subexpression has three advantages:

- The overall structure of the abbreviated expression becomes clearer.
- Duplicate subexpressions are computed only once.
- When writing out code for the abbreviations with `WriteSquaredME`, the abbreviations are sorted into categories depending on their dependence on kinematical variables and are thus computed only as often as necessary.

The combined effect of the latter two points can easily lead to a speed-up by a factor 3.

## 4.7 Resuming Previous Sessions

Loading a list of abbreviations from a previous *FormCalc* session does not by itself mean that `CalcFeynAmp` or `Abbreviate` will use them in subsequent calculations. To this end they must be registered with the system. There are two functions for this.

<code>RegisterAbbr [abbr]</code>	register a list of abbreviations
<code>RegisterSubexpr [subexpr]</code>	register a list of subexpressions

`RegisterAbbr` registers a list of abbreviations, e.g. the output of `Abbr []` in a previous session, such that future invocations of `CalcFeynAmp` will make use of them. Note that abbreviations introduced for different processes are in general not compatible.

`RegisterSubexpr` registers a list of subexpressions, e.g. the output of `Subexpr []` in a previous session, such that future invocations of `Abbreviate` will make use of them.

For long-running calculations, the `Keep` function is helpful to store intermediate expressions, such that the calculation can be resumed after a crash. As a side effect, the intermediate

results can be inspected easily, even while a batch job is in progress.

<code>Keep[expr, name, path]</code>	loads <i>path/name.m</i> if it exists, otherwise evaluates <i>expr</i> and stores the result (together with the output of <code>Abbr[]</code> and <code>Subexpr[]</code> ) in that file. <i>path</i> is optional and defaults to <code>\$KeepDir</code>
<code>Keep[expr, name]</code>	same as <code>Keep[expr, name, \$KeepDir]</code>
<code>Keep[lhs = rhs]</code>	same as <code>lhs = Keep[rhs, "lhs"]</code>
<code>\$KeepDir</code>	the default directory for storing intermediate expressions

`Keep` has two basic arguments: a file (path and name) and an expression. If the file exists, it is loaded. If not, the expression is evaluated and the results stored in the file, thus creating a checkpoint. If the calculation crashes, it suffices to restart the very same program, which will then load all parts of the calculation that have been completed and resume at the point it left off.

The syntax of `Keep` is constructed so as to make adding it to existing programs is as painless as possible. For example, a statement like

```
amps = CalcFeynAmp[...]
```

simply becomes

```
Keep[amps = CalcFeynAmp[...]]
```

Needless to say, this logic fails to work if symbols are being re-assigned, i.e. appear more than once on the left-hand side, as in

```
Keep[amps = CalcFeynAmp[virt]]
Keep[amps = Join[amps, CalcFeynAmp[counter]]
```

Due to the first `Keep` statement, the second will always find the file `keep/amps.m` and never execute the computation of the counter terms.

And there are other ways to confuse the system: mixing intermediate results from different calculations, changing flags out of sync with the intermediate results, etc. In case of doubt, i.e. if results seem suspicious, remove all intermediate files and re-do the calculation from scratch.

## 4.8 Fermionic Matrix Elements

When `FermionChains -> Chiral` or `VA` is chosen, an amplitude involving external fermions will contain `DiracChains`, abbreviated as `Fi`, e.g.

```
F1 -> DiracChain[Spinor[k[2], ME, -1], 6, Lor[1], Spinor[k[1], ME, 1]] *
      DiracChain[Spinor[k[3], MT, 1], 6, Lor[1], Spinor[k[4], MT, -1]]
```

In physical observables such as the cross-section, where only the square of the amplitude or interference terms can enter, these spinor chains can be evaluated without reference to a concrete representation for the spinors. The point is that in terms like  $|\mathcal{M}|^2$  or  $2 \operatorname{Re}(\mathcal{M}_1 \mathcal{M}_2^*)$  only products  $(\bar{F}i Fj)$  of spinor chains appear and these can be calculated using the density matrix for spinors

$$\{u_\lambda(p)\bar{u}_\lambda(p), v_\lambda(p)\bar{v}_\lambda(p)\} = \begin{cases} \frac{1}{2}(1 \pm \lambda\gamma_5)\not{s} & \text{for massless fermions}^\dagger \\ \frac{1}{2}(1 + \lambda\gamma_5\not{s})(\not{p} \pm m) & \text{for massive fermions} \end{cases}$$

where  $\lambda = \pm$  and  $s$  is the helicity reference vector corresponding to the momentum  $p$ .  $s$  is the unit vector in the direction of the spin axis in the particle's rest frame, boosted into the CMS. It is identical to the longitudinal polarization vector of a vector boson, and fulfills  $s \cdot p = 0$  and  $s^2 = -1$ .

In the unpolarized case the  $\lambda$ -dependent part adds up to zero, so the projectors become

$$\sum_{\lambda=\pm} u_\lambda(p)\bar{u}_\lambda(p) = \not{p} + m, \quad \sum_{\lambda=\pm} v_\lambda(p)\bar{v}_\lambda(p) = \not{p} - m.$$

Technically, one can use the same formula as in the polarized case by putting  $\lambda = 0$  and multiplying the result by 2 for each external fermion.

*FormCalc* supplies the function `HelicityME` to calculate these so-called helicity matrix elements  $(\bar{F}i Fj)$ .

---

<sup>†</sup>In the limit  $E \gg m$  the vector  $s$  becomes increasingly parallel to  $p$ , i.e.  $s \sim p/m$ , hence

$$\left. \begin{aligned} \not{s}(\not{p} + m) &= p^2 + m\not{s} = m(\not{s} + m) \\ \not{s}(\not{p} - m) &= p^2 - m\not{s} = -m(\not{s} - m) \end{aligned} \right\} \Rightarrow (1 + \lambda\gamma_5\not{s})(\not{p} \pm m) \xrightarrow{E \gg m} \left(1 + \lambda\gamma_5 \frac{\not{p}}{m}\right)(\not{p} \pm m) = (1 \pm \lambda\gamma_5)\not{p}.$$

<code>HelicityME[<math>\mathcal{M}_1, \mathcal{M}_2^*</math>]</code>	calculate the helicity matrix elements for all combinations of $F_n$ appearing in $\mathcal{M}_1$ and $\mathcal{M}_2^*$ , where, as the $*$ indicates, $\mathcal{M}_2$ is the part to be complex conjugated
<code>All</code>	(used as an argument of <code>HelicityME</code> ;) instead of selecting the $F$ s which appear in an expression, simply take all $F$ s currently in the abbreviations
<code>Mat[<math>F_i, F_j</math>]</code>	the helicity matrix element resulting from the spinor chains in $F_i$ and $F_j$
<code>Hel[<math>n</math>]</code>	the helicity $\lambda_n$ of the $n$ th external particle
<code>s[<math>n</math>]</code>	the helicity reference vector of the $n$ th external particle

Note that `HelicityME` does not calculate the full expression  $\mathcal{M}_1\mathcal{M}_2^*$ , but only the combinations of  $F$ s that appear in this product. These are then called `Mat[ $F_i, F_j$ ]` and depend on the helicities and helicity reference vectors of the external particles, `Hel[ $n$ ]` and `s[ $n$ ]`. See Sect. 4.10 on how to put together the complete expression  $\mathcal{M}_1\mathcal{M}_2^*$ .

If possible, specific values for the `Hel[ $n$ ]` should be fixed in advance, since that can dramatically speed up the calculation and also lead to (much) more compact results. For example, as mentioned before, unpolarized matrix elements can be obtained by putting the helicities  $\lambda_n = 0$  and multiplying by 2 for each external fermion. Therefore, for calculating only the unpolarized amplitude, one could use

```
_Hel = 0;
mat = HelicityME[...]
```

which is generally much faster than `HelicityME[...]` /. `Hel[_] -> 0`. (Don't forget that the matrix elements obtained in this way have yet to be multiplied by 2 for each external fermion.)

*Note:* `HelicityME` uses internal definitions set up by `CalcFeynAmp`. It is therefore not advisable to mix the evaluation of different processes. For instance, wrong results can be expected if one uses `HelicityME` on results from process  $A$  after having computed amplitudes from process  $B$  with `CalcFeynAmp`. Since this requires at least one invocation of `ClearProcess`, though, it is unlikely to happen accidentally.

<i>option</i>	<i>default value</i>	
Dimension	4	the dimension to compute in
MomElim	(taken from CalcFeynAmp)	how to apply momentum conservation
DotExpand	(taken from CalcFeynAmp)	whether to expand terms collected for momentum elimination
RetainFile	False	whether to retain the temporary <i>FORM</i> command file
EditCode	False	whether to display the <i>FORM</i> code in an editor before sending it to <i>FORM</i>

Dimension is used as in CalcFeynAmp. Only the value 0 has the effect of actually computing in  $D$  dimensions, however, since for  $D$  and 4 the limit  $D \rightarrow 4$  has already been taken in CalcFeynAmp. The dimensional dependence of the result is expressed through Dminus4 and Dminus4Eps, where the latter represents the Dminus4 arising from the contraction of Levi-Civita tensors. For testing and comparison, the default equivalence  $\text{Dminus4Eps} = \text{Dminus4}$  can be unset.

The options MomElim, DotExpand, RetainFile, and EditCode are used in the same way as for CalcFeynAmp, see page 13.

## 4.9 Colour Matrix Elements

Diagrams involving quarks or gluons usually<sup>‡</sup> contain objects from the  $SU(N)$  algebra. These are simplified by CalcFeynAmp using the Cvitanovic algorithm [Cv76] in an extended version of the implementation in [Ve96]. The idea is to transform all  $SU(N)$  objects to products of generators  $T_{ij}^a$  which are generically denoted by SUNT in *FormCalc*. In the output, only two types of objects can appear:

- Chains (products) of generators with external colour indices; these are denoted by  $\text{SUNT}[a, b, \dots, i, j] = (T^a T^b \dots)_{ij}$  where  $i$  and  $j$  are the external colour indices and the  $a, b, \dots$  are the indices of external gluons. This notation includes also the identity in colour space as the special case with no external gluons:  $\delta_{ij} = \text{SUNT}[i, j]$ .
- Traces over products of generators; these are denoted by  $\text{SUNT}[a, b, \dots, 0, 0] = \text{Tr}(T^a T^b \dots)$ .

---

<sup>‡</sup>Diagrams generated with the SM.mod model file contain no  $SU(N)$  objects since in the electroweak sector colour can be taken care of by a trivial factor 3 for each quark loop.

The situation is much the same as with fermionic structures: just as an amplitude contains open spinor chains if external fermions are involved, it also contains SUNTs if external quarks or gluons are involved.

For the SUNT objects in the output, *FormCalc* introduces abbreviations of the type  $\text{SUN}n$ . These abbreviations can easily be evaluated further if one computes the squared amplitude, because then the external lines close and the Cvitanovic algorithm yields a simple number for each combination of  $\text{SUN}i$  and  $\text{SUN}j$ . (One can think of the squared amplitude being decomposed into parts, each of which is multiplied by a different colour factor.) But this is precisely the idea of helicity matrix elements applied to the  $\text{SU}(N)$  case!

Because of this close analogy, the combinations of  $\text{SUN}i$  and  $\text{SUN}j$  are called colour matrix elements in *FormCalc* and are written accordingly as  $\text{Mat}[\text{SUN}i, \text{SUN}j]$ . The function which computes them is *ColourME*. It is invoked just like *HelicityME*.

$\text{ColourME}[\mathcal{M}_1, \mathcal{M}_2^*]$	calculate the colour matrix elements for all combinations of $\text{SUN}n$ appearing in $\mathcal{M}_1$ and $\mathcal{M}_2^*$ , where, as the $*$ indicates, $\mathcal{M}_2$ is the part to be complex conjugated
All	(used as an argument of <i>ColourME</i> ;) instead of selecting the SUNs which appear in an expression, simply take all SUNs currently in abbreviations
$\text{Mat}[\text{SUN}i, \text{SUN}j]$	the colour matrix element resulting from the $\text{SU}(N)$ objects $\text{SUN}i$ and $\text{SUN}j$

The core function behind *ColourME* can also be used directly to simplify colour structures.

$\text{ColourSimplify}[expr]$	simplify colour objects in <i>expr</i>
$\text{ColourSimplify}[expr, conj]$	simplifies the colour objects in $(expr\ conj^*)$

Furthermore, *FormCalc* implements a special case of *FeynArts*'s *DiagramGrouping* function in *ColourGrouping*, which groups Feynman diagrams according to their colour structures. The correct grouping can only be done with fully simplified colour structures, which is why this function is part of *FormCalc*, not *FeynArts*.

$\text{ColourGrouping}[ins]$	group the inserted topologies (output of <i>InsertFields</i> ) according to their colour structures
------------------------------	---



### 4.10 Putting together the Squared Amplitude

Now that CalcFeynAmp has calculated the amplitudes and HelicityME and ColourME have produced the helicity and colour matrix elements, the remaining step is to piece together the squared matrix element  $|\mathcal{M}|^2$ , or more generally products like  $\mathcal{M}_1\mathcal{M}_2^*$ . For example,  $\mathcal{M}_1$  could be the sum of the one-loop diagrams and  $\mathcal{M}_2$  the sum of the tree diagrams.

This is non-trivial only if there are helicity or colour matrix elements  $\text{Mat}[i, j]$  which have to be put in the right places. Specifically, if  $\mathcal{M}_1$  and  $\mathcal{M}_2$  are written in the form

$$\begin{aligned}\mathcal{M}_1 &= a_{11} \text{F1 SUN1} + a_{21} \text{F2 SUN1} + \dots = \sum_{ij} a_{ij} \text{Fi SUNj} \quad \text{and} \\ \mathcal{M}_2 &= b_{11} \text{F1 SUN1} + b_{21} \text{F2 SUN1} + \dots = \sum_{ij} b_{ij} \text{Fi SUNj},\end{aligned}$$

$\mathcal{M}_1\mathcal{M}_2^*$  becomes

$$\begin{aligned}\mathcal{M}_1\mathcal{M}_2^* &= a_{11}b_{11}^* \text{Mat}[\text{F1}, \text{F1}] \text{Mat}[\text{SUN1}, \text{SUN1}] + \\ &\quad a_{21}b_{11}^* \text{Mat}[\text{F2}, \text{F1}] \text{Mat}[\text{SUN1}, \text{SUN1}] + \dots \\ &= \sum_{ijkl} a_{ik}b_{j\ell}^* \text{Mat}[\text{Fi}, \text{Fj}] \text{Mat}[\text{SUNk}, \text{SUN}\ell].\end{aligned}$$

The coefficients  $a_{ik}$  and  $b_{j\ell}$  are known as form factors. For efficiency, they are usually computed separately in the numerical evaluation, so that the final expression for the squared matrix element is easily summed up e.g. in Fortran as

```

do 1 i = 1, (# of Fs in  $\mathcal{M}_1$ )
do 1 j = 1, (# of Fs in  $\mathcal{M}_2$ )
do 1 k = 1, (# of SUNs in  $\mathcal{M}_1$ )
do 1 l = 1, (# of SUNs in  $\mathcal{M}_2$ )
    result = result + a(i,k)*DCONJG(b(j,l))*MatF(i,j)*MatSUN(k,l)
1    continue

```

While this is arguably the most economic way to evaluate a squared amplitude numerically, it is also possible to directly obtain the squared matrix element as a *Mathematica* expression. The function which does this is SquaredME.

`SquaredME[ $\mathcal{M}_1, \mathcal{M}_2^*$ ]` calculates  $\mathcal{M}_1\mathcal{M}_2^*$ , taking care to put the  $\text{Mat}[i, j]$  in the right places

SquaredME is called in much the same way as HelicityME. Because of the number of terms that are generated, this function is most useful only for rather small amplitudes.

SquaredME does not insert the actual values for the  $\text{Mat}[i, j]$ . This can easily be done later by applying the output of HelicityME and ColourME, which are lists of rules substituting the

$\text{Mat}[i, j]$  by their values. That is to say, SquaredME and HelicityME or ColourME perform complementary tasks: SquaredME builds up the squared amplitude in terms of the  $\text{Mat}[i, j]$  whereas HelicityME and ColourME calculate the  $\text{Mat}[i, j]$ .

#### 4.11 Polarization Sums

In the presence of external gauge bosons, the output of SquaredME will still contain polarization vectors (in general implicitly, i.e. through the abbreviations). For unpolarized gauge bosons, the latter can be eliminated by means of the identities

$$\sum_{\lambda=1}^3 \varepsilon_{\mu}^*(k, \lambda) \varepsilon_{\nu}(k, \lambda) = -g_{\mu\nu} + \frac{k_{\mu} k_{\nu}}{m^2} \quad \text{for massive particles,}$$

$$\sum_{\lambda=1}^2 \varepsilon_{\mu}^*(k, \lambda) \varepsilon_{\nu}(k, \lambda) = -g_{\mu\nu} - \frac{\eta^2 k_{\mu} k_{\nu}}{(\eta \cdot k)^2} + \frac{\eta_{\mu} k_{\nu} + \eta_{\nu} k_{\mu}}{\eta \cdot k} \quad \text{for massless particles.}$$

In the massless case the polarization sum is gauge dependent and  $\eta$  is an external four-vector which fulfills  $\eta \cdot \varepsilon = 0$  and  $\eta \cdot k \neq 0$ . For a gauge-invariant quantity, the  $\eta$ -dependence should ultimately cancel.

*FormCalc* provides the function `PolarizationSum` to apply the above identities.

`PolarizationSum[expr]`    sums *expr* over the polarizations of external gauge bosons

It is assumed that *expr* is the squared amplitude into which the helicity matrix elements have already been inserted. Alternatively, *expr* may also be given as an amplitude directly, in which case `PolarizationSum` will first invoke `SquaredME` and `HelicityME` (with `_Hel = 0`) to obtain the squared amplitude. `PolarizationSum` cannot simplify Weyl chains, such as `CalcFeynAmp` introduces with `FermionChains -> Weyl` (the default).

<i>option</i>	<i>default value</i>	
SumLegs	All	which external legs to include in the polarization sum
Dimension	4	the dimension to compute in
GaugeTerms	True	whether to retain $\eta$ -dependent terms
MomElim	(taken from CalcFeynAmp)	how to apply momentum conservation
DotExpand	(taken from CalcFeynAmp)	whether to expand terms collected for momentum elimination
NoBracket	(taken from CalcFeynAmp)	symbols not to be included in the bracketing in FORM
RetainFile	False	whether to retain the temporary <i>FORM</i> command file
EditCode	False	whether to display the <i>FORM</i> code in an editor before sending it to <i>FORM</i>

SumLegs allows to restrict the polarization sum to fewer than all external vector bosons. For example, SumLegs  $\rightarrow \{3, 4\}$  sums only vector bosons on legs 3 and 4.

Dimension is used as in CalcFeynAmp. Only the value 0 has the effect of actually computing in  $D$  dimensions, however, since for  $D$  and 4 the limit  $D \rightarrow 4$  has already been taken in CalcFeynAmp. The dimensional dependence of the result is expressed through Dminus4 and Dminus4Eps, where the latter represents the Dminus4 arising from the contraction of Levi-Civita tensors. For testing and comparison, the default equivalence Dminus4Eps = Dminus4 can be unset.

GaugeTerms retains terms containing the gauge-dependent auxiliary vector  $\eta$ . More precisely, the  $\eta$ -terms are actually introduced at first, to let potential cancellations of  $\eta$ 's in the numerator against the denominator occur, but set to zero later.

The options MomElim, DotExpand, NoBracket, RetainFile, and EditCode are used in the same way as for CalcFeynAmp, see page 13.

*Note:* PolarizationSum uses internal definitions set up by CalcFeynAmp. It is therefore not advisable to mix the evaluation of different processes. For instance, wrong results can be expected if one uses PolarizationSum on results from process  $A$  after having computed amplitudes from process  $B$  with CalcFeynAmp. Since this requires at least one invocation of ClearProcess, though, it is unlikely to happen accidentally.

### 4.12 Analytic Unsquared Amplitudes

The ‘smallest’ object appearing in the output of `CalcFeynAmp` is a four-vector, i.e. *FormCalc* does not normally go into components. Those are usually inserted only in the numerical part. This has advantages: for example, the analytical expression does not reflect a particular phase-space parameterization.

One can also obtain an analytic expression in terms of kinematic invariants (but no four-vectors) by squaring the amplitude and computing the polarization sums, as outlined above. This has the advantage of being independent of the representation of the spinors and vectors, but of course the size of the expression is significantly increased by squaring.

As a third alternative, one can obtain an analytical expression for the unsquared amplitude. This requires to go into components, however.

To this end one has to load the extra package `VecSet`:

```
<< FormCalc 'tools' 'VecSet'
```

and for each external vector invoke the function `VecSet`, which has the same syntax as its Fortran namesake, e.g.

```
VecSet[1, m1, p1, {0, 0, 1}]
```

<code>VecSet[n, m, p, {e<sub>x</sub>, e<sub>y</sub>, e<sub>z</sub>}]</code>	set the momentum, polarization vectors, and spinors for particle $n$ with mass $m$ and three-momentum $\tilde{p} = p \{e_x, e_y, e_z\}$
---	---

The amplitude is then evaluated with the function `ToComponents`, e.g.

```
ToComponents[amp, "+--"]
```

This delivers an expression in terms of the phase-space parameters used in `VecSet`.

<code>ToComponents[amp, "p<sub>1</sub>p<sub>2</sub>... p<sub>n</sub>"]</code>	
<code>ToComponents[amp, {p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>}]</code>	evaluate <i>amp</i> by substituting four-vectors by their component-wise representation using external polarizations $p_1, \dots, p_n$

`ToComponents[amp, pol]` actually plugs the components of the four-vectors and spinors into the amplitude *amp*. The external polarizations *pol* can be given either as a string with elements +, -, 0 for right-handed, left-handed, and longitudinal polarization, or as a list of integers +1, -1, 0.

### 4.13 Checking Ultraviolet Finiteness

One way of checking ultraviolet finiteness is to replace the one-loop integrals by their divergent parts and see if the coefficient of the divergence adds up to zero. The function `UVDivergentPart` takes the  $1/(D-4)$ -term of each one-loop integral. This means that non-divergent integrals are set to zero.

<code>UVDivergentPart[expr]</code>	replace all loop integrals in <i>expr</i> by their UV-divergent part
<code>UVSeries[expr]</code>	similar to <code>UVDivergentPart</code> , but perform a series expansion in <code>Dminus4</code>
<code>Divergence</code>	a symbol representing the dimensionally regularized divergence $\Delta = 2/(4-D)$
<code>Dminus4</code>	a symbol representing $D-4$

To assert UV finiteness of an expression, one can check that the following expression is true:

```
uvcheck = UVDivergentPart[expr] //Simplify;
FreeQ[uvcheck, Divergence]
```

Note that models may have ‘hidden’ parameter relations, e.g.  $CW2 = MW2/MZ2$  in the Standard Model, which need to be inserted in order to find analytical cancellation of divergences.

The UV-divergent integrals are

$$\begin{aligned}
A_0(m^2) &= m^2 \Delta + \mathcal{O}(1), & C_{00} &= \frac{\Delta}{4} + \mathcal{O}(1), \\
A_{00}(m^2) &= \frac{m^4}{4} \Delta + \mathcal{O}(1), & C_{00i} &= -\frac{\Delta}{12} + \mathcal{O}(1), \\
B_0 &= \Delta + \mathcal{O}(1), & C_{00ii} &= \frac{\Delta}{24} + \mathcal{O}(1), \\
B_1 &= -\frac{\Delta}{2} + \mathcal{O}(1), & C_{00ij} &= \frac{\Delta}{48} + \mathcal{O}(1), \\
B_{00}(p^2, m_1^2, m_2^2) &= \left( \frac{m_1^2 + m_2^2}{4} - \frac{p^2}{12} \right) \Delta + \mathcal{O}(1), & D_{0000} &= \frac{\Delta}{24} + \mathcal{O}(1), \\
\frac{\partial B_{00}(p^2, m_1^2, m_2^2)}{\partial p^2} &= -\frac{\Delta}{12} + \mathcal{O}(1), & D_{0000i} &= -\frac{\Delta}{96} + \mathcal{O}(1), \\
B_{11} &= \frac{\Delta}{3} + \mathcal{O}(1), \\
C_{0000}(p_1^2, p_2^2, p_3^2, m_1^2, m_2^2, m_3^2) &= \left( \frac{m_1^2 + m_2^2 + m_3^2}{24} - \frac{p_1^2 + p_2^2 + p_3^2}{96} \right) \Delta + \mathcal{O}(1).
\end{aligned}$$

Alternatively, UV finiteness may be checked numerically by varying the parameters that regularize the infinity in the loop integrals,  $\mu$  and  $\Delta$ , on which the final result must not depend. This is of course limited to the achievable numerical precision.

#### 4.14 Useful Functions

<code>DenCollect[expr]</code>	collects terms in <i>expr</i> whose denominators are identical up to a numerical constant
<code>DenCollect[expr, wrap]</code>	additionally applies <i>wrap</i> to the collected numerators
<code>TagCollect[expr, tag, wrap]</code>	apply <i>wrap</i> to the part of <i>expr</i> tagged by <i>tag</i>
<code>TermCollect[expr]</code>	combines terms with common factors, i.e. does $ab + ac + d \rightarrow a(b + c) + d$
<code>Pool[expr]</code>	same as <code>TermCollect</code> but different method
<code>Pool[expr, wrap]</code>	additionally applies <i>wrap</i> to the $b + c$ part
<code>DotSimplify[f<sub>1</sub>, f<sub>2</sub>][expr]</code>	simplify <i>expr</i> with $f_1$ if free of any <code>NoBracket</code> items, else with $f_2$
<code>OnSize[n<sub>1</sub>, f<sub>1</sub>, n<sub>2</sub>, f<sub>2</sub>, ..., f<sub>def</sub>][expr]</code>	returns $f_1[expr]$ if <code>LeafCount[expr] &lt; n<sub>1</sub></code> , $f_2[expr]$ if <code>LeafCount[expr] &lt; n<sub>2</sub></code> , etc., and $f_{def}[expr]$ if the expression is still larger
<code>ApplyUnitarity[expr, U, d, s]</code>	simplify <i>expr</i> by exploiting the unitarity of the $d$ -dimensional matrix $U$ . The optional argument <i>s</i> specifies the simplification function to use internally (default: <code>FullSimplify</code> )
<code>ExpandSums[expr]</code>	turns all pieces of <i>expr</i> multiplied with <code>SumOver</code> into an actual sum

`DenCollect` works essentially like *Mathematica*'s `Collect`, except that it takes the denominators appearing in the expression to collect for. Note that this refers to the 'real' denominators, i.e. parts of the expression with negative powers, not to the symbol `Den[p, m]` which stands for propagator denominators (the latter can easily be collected with `Collect`).

`TagCollect` collects an expression with respect to powers of a tag and applies a function to the term linear in the tag. This function is typically used to apply a function to the tagged part of an expression, such as is possible with the `MultiplyDiagrams` function.

`TermCollect` and `Pool` combine terms with common factors. Unlike `Factor`, they look at the terms pairwise and can thus do  $ab + ac + d \rightarrow a(b + c) + d$  fast. Unlike `Simplify`, they do not modify  $b$  and  $c$ .

`DotSimplify` simplifies an expression with two different functions depending on whether it contains any of the objects listed in the `NoBracket` option (i.e. during the execution of `CalcFeynAmp` or `PolarizationSum`).

The `OnSize` function is similar to the `Switch` statement, but for the size of the expression. This can be useful when simplifying expressions because `Simplify` (even `FullSimplify`) is fast and efficient on short expressions, but tends to be slow on large ones. For example,

```
OnSize[100, FullSimplify, 500, Simplify, TermCollect]
```

would apply `FullSimplify` if the expression's leaf count is less than 100, `Simplify` if it is between 100 and 500, and `DenCollect` above.

`ApplyUnitarity` exploits unitarity of a given matrix to simplify an expression. The two unitarity relations  $UU^\dagger = \mathbb{1}$  or  $\sum_{i=1}^d U_{ij}U_{kj}^* = \delta_{ik}$  and  $U^\dagger U = \mathbb{1}$  or  $\sum_{i=1}^d U_{jk}U_{ji}^* = \delta_{ik}$  are used to substitute sums of  $UU^*$  elements with more than  $\lceil d/2 \rceil$  terms. For example, `ApplyUnitarity[expr, CKM, 3]` might replace  $CKM_{12}CKM_{13}^* + CKM_{22}CKM_{23}^*$  by  $-CKM_{32}CKM_{33}^*$ .

`ExpandSums` turns expressions where index summations are denoted by a `SumOver`  $[i, r]$  multiplied with the part of the expression to be summed over index  $i$ , such as delivered by `CalcFeynAmp`, into ordinary Mathematica sums. `ExpandSums[expr, h]` uses head  $h$  instead of `Plus`.

## 5 Tools for the Numerical Evaluation

The numerical evaluation has to take care of two important issues, and therefore resolves into two conceptual steps:

1. Almost invariably, the numerical evaluation in *Mathematica* itself is too slow. Even though the speed of numerical computations has been improved greatly in *Mathematica* 4, the sheer amount of number crunching e.g. in a scan over parameter space places the numerical evaluation safely in the domain of a compiled language. This makes it necessary to translate the *Mathematica* expressions resulting from `CalcFeynAmp`, `HelicityME`, etc. into a high-level language program.

For *FormCalc*, Fortran has been chosen because of its proven track record for fast and reliable number crunching, the availability of good compilers on all platforms, and the possibility to link with existing code.

This first step of writing out a subroutine from the calculated amplitudes can be implemented in a reasonably general way and is handled by *FormCalc* itself, see Sect. 5.1.

2. What *FormCalc* generates is just the subroutine for computing the squared matrix element. Of course, this subroutine has to be provided with the proper kinematics, the model parameters have to be initialized, etc. In other words, a driver program is required to invoke the generated subroutine. Unfortunately, it is not really possible to

write one single fully automated driver program that fits each and every purpose, or rather, such a program would be difficult to use beyond its designed scope.

The driver programs that come with *FormCalc* together are a powerful tool for computing cross-sections, but moreover they have expressly been written to give an example of how the generated code can be used. Extending or modifying this code for a related purpose should be fairly straightforward.

The following files in the *FormCalc* distribution are used for the numerical evaluation:

drivers/	driver programs for running the generated code:
process.h	the process specification
run.F	the parameters for a particular “run”
partonic.h	the partonic composition of the final result
extra.h	extra user definitions
distrib.h	definitions for distributed computing
makefile.in	the makefile minus the flags set by configure
configure	a script to find out library locations and compiler flags
do	a script to automate computations
drivers/F/	Fortran driver programs for running the generated code:
main.F	the main program (command-line processing)
xsection.F,.h	the code for computing the cross-section
parton.h	setup for a single partonic process
num.h	definitions for the OPP numerator functions
model_*.F,.h	initialization of the model parameters
lumi_*.F	luminosity computation
MtoN.F,.h	the kinematics for a $M \rightarrow N$ process
softphoton.F	the integrals for the soft-photon approximation
util.h	definitions for the functions in <code>util.a</code>
const.h	general constants and regularization parameters
inline.h	inline versions of the util functions
contains.h	inline versions of the util functions
user.h	user definitions, e.g. choice of model, OPP, etc.
decl.h	combined declarations file, included ‘everywhere’
drivers/C/	C driver programs for running the generated code:
num.h	definitions for the OPP numerator functions
model_*.h	declarations of the model parameters
util.h	definitions for the functions in <code>util.a</code>
const.h	general constants and regularization parameters
inline.h	inline versions of the util functions



<code>contains.h</code>	inline versions of the util functions
<code>user.h</code>	user definitions, e.g. choice of model, OPP, etc.
<code>decl.h</code>	combined declarations file, included 'everywhere'
<code>drivers/tools/</code>	helper scripts:
<code>mktm</code>	a script to set up Mathematica interfacing code (internal use)
<code>data</code>	a script for extracting the actual data out of the log files
<code>pnuglot</code>	a script for plotting data files using gnuplot
<code>sfx</code>	a script for making a self-extracting archive
<code>turnoff</code>	a script to turn on or off the evaluation of code modules
<code>submit</code>	a script to submit parallel jobs
<code>tools/</code>	tools for the numerical evaluation:
<code>ReadData.tm</code>	a program for reading data files into <i>Mathematica</i>
<code>reorder.c</code>	a utility to reorganize parameters in data files
<code>ScanGraphics.m</code>	special <i>Mathematica</i> graphics functions for parameter scans
<code>bfunc.m</code>	the explicit expressions for the one- and two-point functions
<code>btensor.m</code>	the explicit tensor reduction for the one- and two-point functions in <i>Mathematica</i>

## 5.1 Generating code

In most cases, the numerical evaluation of an amplitude directly in *Mathematica* is too slow, so the *FormCalc* results need to be converted to a Fortran or C program. The simplest way to do this in *Mathematica* is something like `FortranForm[result] >> file.f`. Unfortunately, this leaves a lot to be done manually.

*FormCalc* contains two much more sophisticated functions for generating code, `WriteSquaredME` and `WriteRenConst`. The philosophy is that the user should not have to modify the generated code. This eliminates the source of many errors, but of course goes well beyond simple translation to code: the code has to be encapsulated (i.e. no loose ends the user has to bother with), and all necessary subsidiary files, such as include files and a makefile, have to be produced. Also, the code has to be chopped into pieces small enough that the compiler will accept them.

Before using `WriteSquaredME` or `WriteRenConst`, a directory must be created for the code and the driver programs copied into this directory. This is done with `SetupCodeDir`:

<code>SetupCodeDir[dir]</code>	create a directory <i>dir</i> and install the driver programs necessary to compile the generated code in this directory
--------------------------------	---

Note that `SetupCodeDir` never removes or overwrites the directory or the driver programs, so a code directory, once created, stays put. The `Drivers` option can be used to load customized driver programs from a local directory.

<i>Option</i>	<i>default value</i>	
<code>Drivers</code>	<code>"drivers"</code>	a directory containing customized versions of the driver programs which take precedence over the default ones in <code>\$DriversDir</code>

To illustrate the concept, consider generating code for a particular scattering process. The first time, no customized drivers exist and hence *FormCalc* copies the default drivers into the code directory. The user then modifies, say, `process.h`. To preserve the changes, he copies the file to a local directory, e.g. `drivers`, and specifies this directory with the `Drivers` option. Now even if the code directory is deleted, generating the code anew will recover all modifications because the customized version of `process.h` in `drivers` supersedes the default `process.h` in `$DriversDir`.

The function `WriteSquaredME` assumes that there is a tree-level amplitude  $\mathcal{M}_{\text{tree}}$  and a one-loop amplitude  $\mathcal{M}_{1\text{-loop}}$  and writes out a subroutine called `SquaredME` which computes the two expressions  $|\mathcal{M}_{\text{tree}}|^2$  and  $2 \text{Re } \mathcal{M}_{\text{tree}}^* \mathcal{M}_{1\text{-loop}}$ . When there is no tree-level contribution, 0 and  $|\mathcal{M}_{1\text{-loop}}|^2$  are returned, and when there is no one-loop contribution,  $|\mathcal{M}_{\text{tree}}|^2$  and 0 are returned.

`WriteSquaredME[tree, loop, mat, abbr, ..., dir]`

generate code for a subroutine `SquaredME` in the directory *dir*; this subroutine computes the squared matrix element composed of the amplitudes *tree* (for the tree-level part) and *loop* (for the one-loop part)

The *tree* and *loop* arguments are results of `CalcFeynAmp` and are trailed by all other objects necessary for the computation of the squared matrix element, like helicity matrix elements, colour matrix elements, and the abbreviations. An empty list, `{}`, is used to indicate that the *tree* or *loop* argument is missing. Since `WriteSquaredME` has a rather complicated invocation, it is perhaps best to give an example:

```
born = CalcFeynAmp[bornamps];
self = CalcFeynAmp[selfamps];
vert = CalcFeynAmp[vertamps];
box = CalcFeynAmp[boxamps];
col = ColourME[All, born];
```

```
dir = SetupCodeDir["fortran"];
WriteSquaredME[born, {self, vert, box}, col, Abbr[], dir]
```

The *tree* and *loop* arguments of `WriteSquaredME` have the somewhat unorthodox feature that the naming of the code modules depends on the variable names chosen for the components of the amplitude (like *born*, *self*, etc., in the example above). That is, `WriteSquaredME` tries to name the code modules like the variables, e.g. the amplitude contained in the variable *self* will be written to files of the form *self\*.F*. This is only possible if the variable names are plain symbols; in all other cases, the modules are named *Tree1.F*, *Tree2.F*, or *Loop1.F*, *Loop2.F*, etc.

The following options can be used with `WriteSquaredME`.

<i>option</i>	<i>default value</i>	
<code>ExtraRules</code>	<code>{}</code>	extra rules to be applied to the expressions before the loop integrals are abbreviated
<code>TreeSquare</code>	<code>True</code>	whether the square of the tree-level amplitude is added to the result
<code>LoopSquare</code>	<code>False</code>	whether the square of the one-loop amplitude is added to the result
<code>Folder</code>	<code>"squaredme"</code>	the subdirectory of the code directory into which the generated code is written
<code>FilePrefix</code>	<code>""</code>	a string prepended to the filenames of the generated code
<code>SymbolPrefix</code>	<code>""</code>	a string prepended to global symbols to prevent collision of names when more than one process is linked
<code>FileHeader</code>	<code>"#if 0\n * %f\n * %d\n * generated by FormCalc m.n%t\n #endif\n\n"</code>	the file header
<code>SubroutineIncludes</code>	<code>FileIncludes</code>	per-subroutine <code>#include</code> statements
<code>FileIncludes</code>	<code>FileIncludes</code>	per-file <code>#include</code> statements
	<code>{"#include \"decl.h\", "#include \"inline.h\"\n", "#include \"contains.h\"\n"}</code>	

`ExtraRules` specifies additional transformation rules that are applied to the modules of the amplitude before the loop integrals are abbreviated.

`TreeSquare` determines whether the square of the tree-level amplitude is added to the result. This is the case unless one needs to compute only the interference term, usually for testing.

`LoopSquare` determines whether the square of the one-loop amplitude is added to the one-loop result, i.e. whether the `SquaredME` subroutine shall compute  $2 \operatorname{Re} \mathcal{M}_{\text{tree}}^* \mathcal{M}_{1\text{-loop}} + |\mathcal{M}_{1\text{-loop}}|^2$  rather than only  $2 \operatorname{Re} \mathcal{M}_{\text{tree}}^* \mathcal{M}_{1\text{-loop}}$ . Without the  $|\mathcal{M}_{1\text{-loop}}|^2$  term (which completes the square  $|\mathcal{M}_{\text{tree}} + \mathcal{M}_{1\text{-loop}}|^2$ ), the one-loop result may become negative if  $\mathcal{M}_{\text{tree}}$  is very small. The contribution of the same order from the two-loop result,  $2 \operatorname{Re} \mathcal{M}_{\text{tree}}^* \mathcal{M}_{2\text{-loop}}$ , is much smaller than  $|\mathcal{M}_{1\text{-loop}}|^2$  in this case so that there is no inconsistency with the order in perturbation theory.

`Folder` specifies the name of the subdirectory relative to the code directory into which the generated code is written. When using this option, the default makefile will usually have to be adapted accordingly.

`FilePrefix` specifies a string which will be prepended to the names of all generated files. This complements the `Folder` option as another way of making files unique.

`SymbolPrefix` is a way of avoiding symbol collisions when more than one process generated by `WriteSquaredME` is linked together. The given string must be acceptable to Fortran or C as it is prepended to all global symbols like the names of subroutines and common blocks.

`FileHeader` specifies a string to be used as file header. This string may contain `%f`, `%d`, and `%t`, which are substituted at file creation by file name, description, and time stamp, respectively.

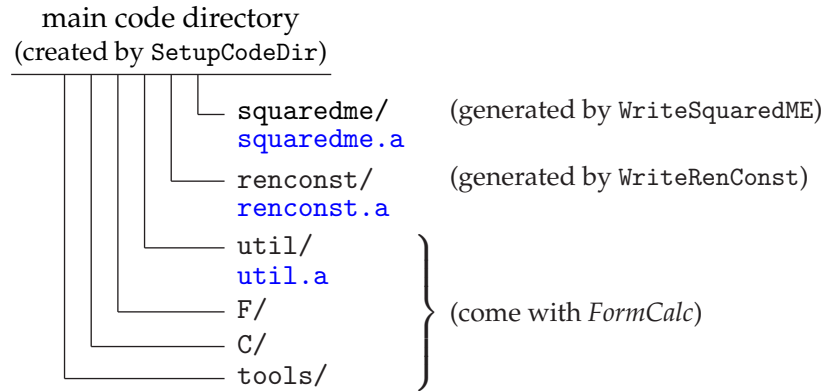
`SubroutineIncludes` gives declarations to be inserted in the declaration section of each subroutine, usually in the form of `#include` statements. In the most general form a list of three strings is given, the first of which included before, the second after local variable declarations, and the third at the end of the routine. It is admissible to provide a list with fewer strings, or just a string (without a list), in which case the unspecified ones remain empty.

`FileIncludes` gives declarations to be inserted at the beginning of each generated file. It accepts the same input as `SubroutineInclude` except that there is no subroutine relative to which the placement can be made. Instead, the first list element goes into the central include file `vars.h`, the other two into the individual code files, after the `#include "vars.h"` statement.

<code>SetLanguage["Fortran"]</code>	set the output language to Fortran (default)
<code>SetLanguage["C"]</code>	set the output language to C

## Libraries and Makefiles

The code is organized in *FormCalc* into a main code directory, which contains the main program and all its prerequisite files, and subsidiary ‘folders’ (subdirectories to the main code directory). The default setup looks like this:



Folders equipped with an own makefile produce a library of the same name, e.g. the makefile in `util/` makes the library `util.a`. These sub-makefiles are orchestrated by the master makefile. Libraries required for the main program are listed in the `LIBS` variable and built automatically by invoking the sub-makefiles:

```
LIBS = squaredme.a renconst.a util.a
```

Note that `configure` overwrites `makefile`, hence ‘permanent’ changes should be made in `makefile.in` since `configure` overwrites `makefile`.

The `util` library is a collection of ancillary routines which currently includes:

- System utilities (log file management),
- Kinematic functions (`Pair`, `Eps`, ...),
- Diagonalization routines (DIAG library [Ha06]),
- Univariate integrators (Gauss, Patterson),
- Multivariate integrators (CUBA library [Ha04]).

The `util.a` library is compiled once when *FormCalc* is installed and then copied to the main code directory, thus avoiding unnecessary compiles.

## Partonic Composition

Partonic processes can be combined in the final result. The number of incoming and outgoing legs must be the same, so that the phase-spaces have identical dimension for integration,

but otherwise each partonic process may have distinct kinematics. It is up to the user to ensure that the combined result makes sense physically.

Technically one invokes `WriteSquaredME` for each partonic process, but with different values for the `Folder` and `SymbolPrefix` option. The `Folder` option chooses different output directories and the `SymbolPrefix` option disambiguates globally visible symbols in the generated code. Both options admit the use of `ProcName`, as in: `Folder -> {"squaredme", ProcName}` which will be substituted by the process name, suitably arranged as file and symbol name. The above folder name might be expanded as `squaredme/Fb21F21Fb33iF33i`.

The actual composition of the final result is determined through the `partonic.h` header in the drivers directory. The default setup, for an only partonic process, is

```
#define NPID 1

*** BEGIN PARTONIC PROCESS #1

#define PID 1
#define PARTON1 -1
#define PARTON2 -1
#include "squaredme/specs.h"
#include "parton.h"

*** END PARTONIC PROCESS #1
```

For more than one partonic process, `NPID` has to be increased appropriately and the lines between `BEGIN` and `END PARTONIC PROCESS` have to be replicated for each partonic process, with the following quantities adapted:

- `PID` is the running number of the partonic process, counting from 1 to `NPID` (though not necessarily in ascending order),
- `PARTON1` and `PARTON2` identify the incoming partons in `lumi_hadron.F` by their PDG code: 0 = gluon, 1 = down, 2 = up, 3 = strange, 4 = charm, 5 = bottom, 6 = top.
- The `specs.h` file from the correct directory (cf. `Folder` option above) has to be included.
- The `parton.h` stays put, i.e. is the same for each partonic process.

### Specifying numerical parameters

Numerical parameters are specified in the model initialization file, e.g. `model_sm.F`, with declarations in `model_sm.h`. Each such initialization file must implement the following three subroutines:

```
subroutine ModelDefaults
```

sets all model parameters to their initial values. This subroutine is invoked before the user gets to set the input parameters.

```
subroutine ModelConstIni(fail)
integer fail
```

initializes the constant model parameters, i.e. the ones that do not depend on  $\sqrt{s}$ , the centre-of-mass energy. If the routine determines that the input parameters yield unphysical or otherwise unacceptable model parameters, it must return a non-zero value in fail.

```
subroutine ModelVarIni(fail, sqrtS)
integer fail
double precision sqrtS
```

is called after ModelConstIni to initialize the model parameters that depend on  $\sqrt{s}$ , typically  $\alpha_s$  and derived quantities. Finally, the

```
subroutine ModelDigest
```

is invoked to print a digest of the model parameters.

Note one detail: some model constants are declared in `model_x.h` with parameter statements, i.e. as numerical constants. WriteSquaredME can optimize the generated code based on the information which symbols are numerical constants to the compiler and which are variables.

The trick is to move the numerical constants to the front of each product so that the compiler can combine these constants into one number, thus reducing the number of multiplications at run time. For example, if WriteSquaredME knows Alfa2 and SW2 to be numerical constants, it will write  $(3*Alfa2)/(8.D0*SW2**2)*(Abb1*cint44*MH2)$  instead of *Mathematica's* default ordering  $(3*Abb1*Alfa2*cint44*MH2)/(8.D0*SW2**2)$ . While this may seem a minor rearrangement, the increase in execution speed can be 30%!

To tell WriteSquaredME which symbols are numerical constants to the compiler, simply assign the symbol a numerical value like in (this statement is near the end of `FormCalc.m`)

```
Scan[ (N[#] = Random[])&, {Alfa, Alfa2, MW, MW2, ...} ]
```

### Details of the code generation

The following excerpt is from a Fortran code generated by WriteSquaredME:

```

subroutine self
implicit none

#include "vars.h"

      Cloop(1) = Cloop(1) +
&      ((3*Alfa2)/(8.D0*SW2**2)*(Abb1*cint44*MH2) +
&      (9*Alfa2)/(8.D0*SW2**2)*(Abb1*cint5*MH2**2) +
&      Alfa2/(4.D0*SW2**2)*(Abb1*cint41*(MH2 + 6*MW2)) +
&      (Alfa2*MW2)/SW2**2*(Abb1*cint43*S) +
&      (Alfa2*MW2)/(2.D0*CW2*SW2**2)*(Abb1*cint46*S) +
&      Alfa2/(8.D0*CW2*SW2**2)*
&      (Abb1*cint45*(6*MW2 + CW2*MH2)) ...

```

Note the use of the C preprocessor in statements like `#include`. Because of this, the code files have the extension `.F` which is recognized by almost all Fortran compilers.

`WriteSquaredME` pulls the calculation of several kinds of objects out of the actual computation of the amplitude. That is, these objects are calculated first and stored in variables, and then the amplitude is computed. Because the replacement of more complex objects by scalar variables shortens the expressions, these objects are referred to as abbreviations.

The following objects are treated as abbreviations:

1. variables replacing some function call:  $\text{Pair}n$  (dot products),  $\text{Eps}n$  (contracted epsilon tensors),  $\text{cint}n$ ,  $\text{iint}n$  (loop integrals),
2. products or sums of variables of type 1:  $\text{Abb}n$ ,  $\text{AbbSum}n$ ,
3. matrix elements:  $\text{MatType}(i, j)$ .

All of these abbreviations except the  $\text{cint}n$  and  $\text{iint}n$  are in fact the ones already introduced by *FormCalc*. These abbreviations share the general feature of being costly in CPU time (in particular the loop integrals), which means that for optimal performance they should be calculated as few times as possible. `WriteSquaredME` splits them into  $2 \times 3$  categories:

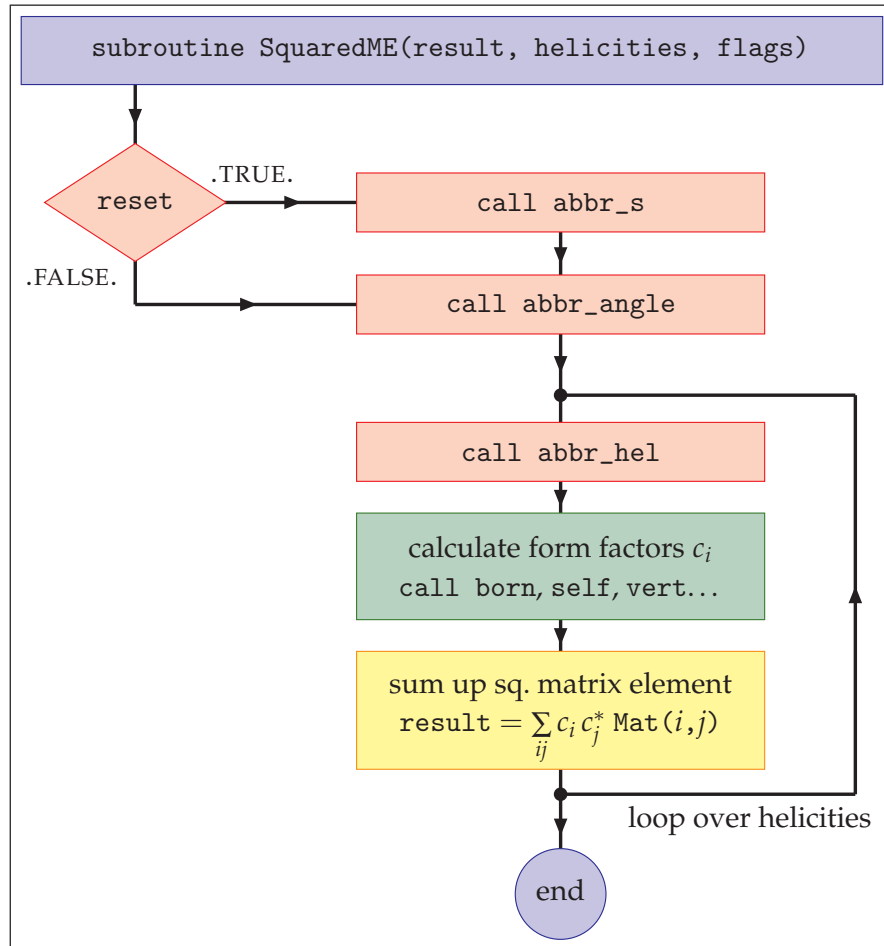
- `abbr0_cat.F` – tree-level, and
- `abbr1_cat.F` – one-loop,
- `abbr_i_s.F` – abbreviations which depend only on  $\sqrt{s}$ ,
- `abbr_i_angle.F` – abbreviations which depend also on other phase-space variables, but not on the helicities, nad



- `abbr_i_hel.F` – helicity-dependent abbreviations.

Needless to say, each of these `abbr_cat` modules is invoked only when necessary.

All modules generated by `WriteSquaredME` are invoked in the proper fashion by the master subroutine `SquaredME`, as shown in the following figure.



As long as one sticks to the default driver programs, one does not have to worry about actually calling `SquaredME`. For certain applications (e.g. Monte Carlo generators) it might nevertheless be attractive to invoke `SquaredME` directly. Its declaration is

```

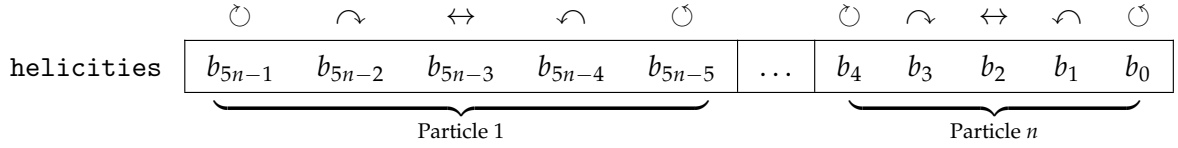
subroutine SquaredME(result, helicities, flags)
double precision result(2)
integer*8 helicities
integer flags

```

The results are returned in `result`, `helicities` encodes the helicities to include, and `flags` is an integer specifying flags for the computation.

The results of SquaredME are  $\text{result}(1) = |\mathcal{M}_{\text{tree}}|^2$  and  $\text{result}(2) = 2 \text{Re } \mathcal{M}_{\text{tree}}^* \mathcal{M}_{1\text{-loop}}$ , except when there is no tree-level contribution, in which case  $\text{result}(1) = 0$  and  $\text{result}(2) = |\mathcal{M}_{1\text{-loop}}|^2$  is returned.

The helicities are encoded bitwise in the integer argument `helicities`. For each external particle, five bits represent, from most to least significant bit, right-circular (spin 2 or 3/2), right-circular (spin 1 or 1/2), longitudinal, left-circular (spin 1 or 1/2) and left-circular (spin 2 or 3/2) polarization:

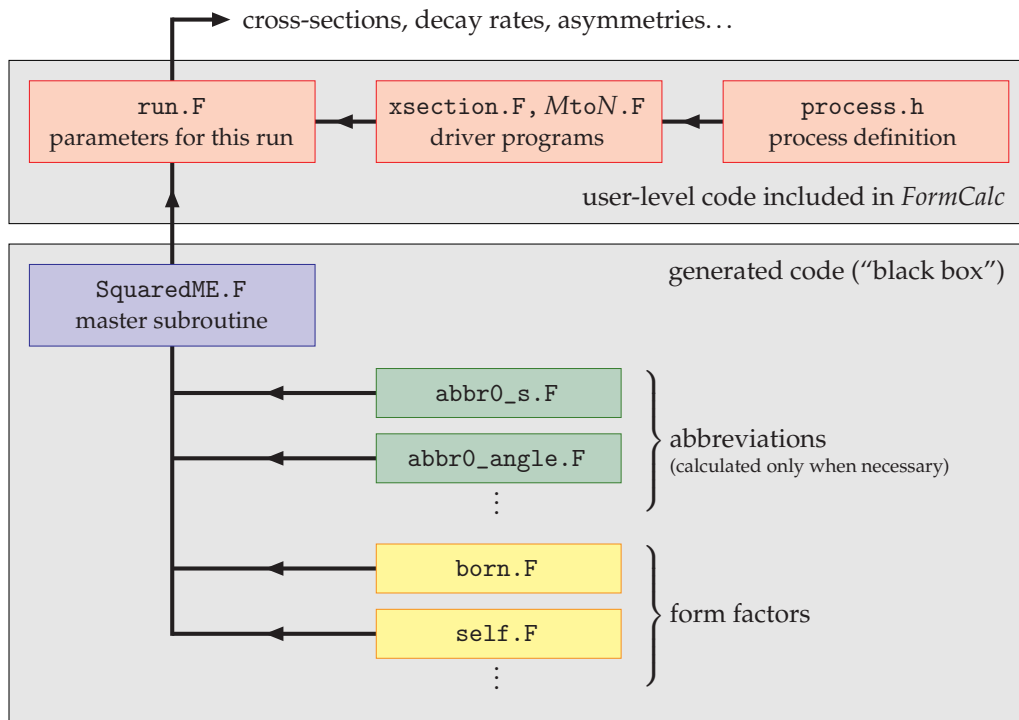


For instance, a left- and right-circularly polarized fermion scattering into an unpolarized two-vector-boson final state would be given by `helicities = 00010 01000 01110 011102 = 7419010`. Some compilers allow binary notation directly as `B'00010 01000 01110 01110'` (this is non-portable, however).

Currently two flags are passed:

- Bit 0 is the ‘reset’ flag: if 1, the abbreviations must be re-calculated completely. The latter is the case when the centre-of-mass energy or the model parameters have changed.
- Bit 1 is the ‘loop’ flag: if 1, the loop corrections are computed. This flag allows e.g. to integrate phase space separately for tree-level and one-loop part.

The following picture shows the default constellation of driver programs and code modules generated by `WriteSquaredME`:



## 5.2 Running the Generated Code

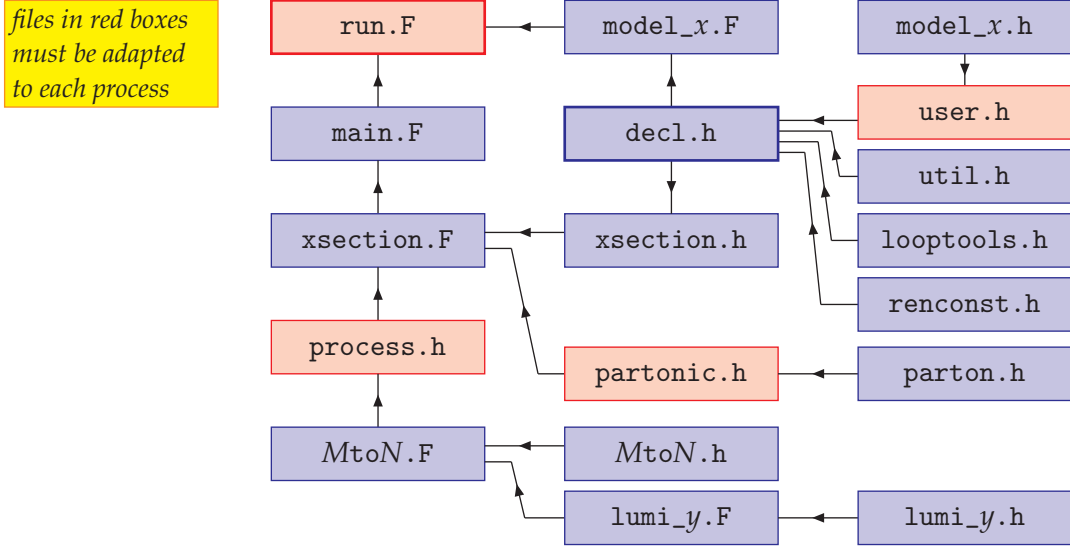
The code produced by `WriteSquaredME` needs in addition a driver program which supplies it with the necessary parameters, kinematics, etc. The default driver program that comes with *FormCalc* is organized in a very modular way and spreads out over several files: Technically, the main file is `run.F`, i.e. this is the file the compiler sees as main program. However, `run.F` is just a clever way of defining parameters at the right places, and the real work is done in other files which are included by `run.F`.

There is almost no cross-talk between different modules which are in that sense ‘universal.’ Also, the main program has been radically slimmed down in *FormCalc* 5. It now only scans the command line and invokes

```
call ProcessIni(...)
call ParameterScan(...)
```

All further action is decoupled from the main program and can easily be called from any application. It is thus straightforward to use *FormCalc*-generated code in own programs.

The distribution of the code over the various include files and their interdependencies is shown in the following figure, where arrows indicate code inclusion via `#include`.



Organizing the code in this seemingly entangled, but highly modular way makes it possible for one program to perform many different tasks simply by setting preprocessor flags. The different modules have the following duties:

- `run.F` defines a “run,” e.g. the ranges over which to scan model parameters,
- `process.h` defines all process-dependent parameters,
- `main.F` does the command-line parsing,
- `xsection.F` contains the kinematics-independent code,  
`xsection.h` contains the declarations for `xsection.F`,
- `partonic.h` determines the partonic composition of the result,  
`parton.h` contains the code for a single partonic process,
- `MtoN.F` contains the kinematics-dependent code for a  $M \rightarrow N$  process,  
`MtoN.h` contains the declarations for `MtoN.F`,
- `model_x.F` (currently one of `model_sm.F`, `model_mssm.F`, `model_thdm.F`) initializes the model parameters,  
`model_x.h` contains the declarations for `model_x.F`,
- `lumi_y.F` (currently one of `lumi_parton.F`, `lumi_hadron.F`, `lumi_photon.F`) calculates the parton luminosity,  
`lumi_y.h` contains the declarations for `lumi_y.F`,
- `util.h` contains the declarations for the functions in the `util` library,
- `looptools.h` is the *LoopTools* include file,

- `renconst.h` declares the renormalization constants (see Sect. 5.4).

In this setup, the choice of parameters is directed by the two files `process.h` and `run.F`, which include one each of

- Kinematics definitions:
  - `1to2.F`,
  - `2to2.F`,
  - `2to3.F`,
- Convolution with PDFs:
  - `lumi_parton.F`,
  - `lumi_hadron.F`,
  - `lumi_photon.F`,
- Model initialization:
  - `model_sm.F`,
  - `model_mssm.F`,
  - `model_thdm.F`.

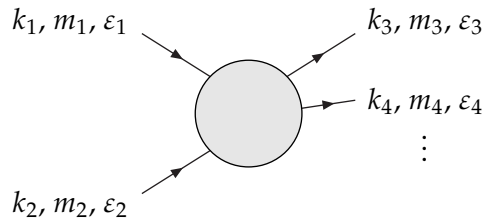
### 5.2.1 Process definition

All process-specific definitions are given in `process.h`. The external particles are declared in the lines

```
#define TYPE $i$   $t_i$ 
#define MASS $i$   $m_i$ 
#define CHARGE $i$   $c_i$ 
```

where each  $t_i$  is one of the symbols SCALAR, FERMION, VECTOR, PHOTON, or GLUON. PHOTON is the same as VECTOR except that longitudinal polarization states are not allowed and GLUON is just an alias for PHOTON.

As in *FormCalc*, the momenta, masses, and polarization vectors are numbered sequentially like in the following figure.

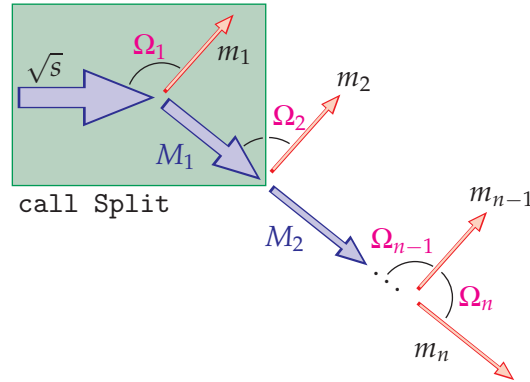


In addition to the external particles, the following items are defined in `process.h`:

- a colour factor (COLOURFACTOR), to account for coloured particles in the initial state,
- a combinatorial factor (IDENTICALFACTOR), to account for identical particles in the final state,
- a wave-function renormalization (WF\_RENORMALIZATION), if a non-onshell renormalization scheme is used,
- whether soft bremsstrahlung shall be added (BREMSSTRAHLUNG), and the maximum energy a soft photon may have (ESOFTMAX) (see Sect. 5.5 for details).

### 5.2.2 Building up phase space

*FormCalc* builds up the  $n$ -particle phase space iteratively by nested integrations over the invariant mass  $M_i$  and solid angle  $\Omega_i$  of each outgoing particle  $i$ . This procedure is encoded in the subroutine `Split`:



Counting the degrees of freedom, there are  $(n - 1)$   $M$ -integrations and  $n$   $\Omega$ -integrations. The corresponding phase-space parameterization is

$$\begin{aligned}
 & \frac{1}{2\sqrt{s}} \int_{m_2+\dots+m_n}^{\sqrt{s}-m_1} dM_1 d\Omega_1 \frac{k_1}{2} \\
 & \times \int_{m_3+\dots+m_n}^{M_1-m_2} dM_2 d\Omega_2 \frac{k_2}{2} \\
 & \times \dots \\
 & \times \int_{m_n}^{M_{n-2}-m_{n-1}} dM_{n-1} d\Omega_{n-1} \frac{k_{n-1}}{2} \\
 & \times \int d\Omega_n \frac{k_n}{2}
 \end{aligned}$$

where  $d\Omega_i = d\cos\theta_i d\varphi_i$ . The particle's momentum  $k_i$  and  $\cos\theta_i$  are given in the respective decay's rest frame. The  $\varphi_1$ -integration is trivial because of axial symmetry. From the practical point of view this looks as follows (this code is taken almost verbatim from *FormCalc*'s 2to3.F):

```

p = 0
ex = 0
ey = 0
ez = 1
minv = sqrtS
msum = MASS3 + MASS4 + MASS5

call Split(5, dble(MASS5),
& p, ex,ey,ez, minv, msum, fac, 0,
& Var(XMREM5), Var(XCOSTH5), Var(TRIVIAL))

call Split(4, dble(MASS4),
& p, ex,ey,ez, minv, msum, fac, 0,
& Var(FIXED), Var(XCOSTH4), Var(XPHI4))

call VecSet(3, dble(MASS3), p, ex,ey,ez)

```

One starts with the initial reference direction in  $(ex, ey, ez)$  and no boost,  $p = 0$ . The available energy is given in `minv` and the sum of external masses in `msum`. The `Split` subroutine is then called  $(n - 1)$  times for an  $n$ -particle final state. The reference direction, the boost, `minv`, and `msum` are automatically adjusted along the way for the respective remaining subsystem and ultimately determine the remaining  $n$ -th vector unambiguously, which is then simply set by `VecSet`.

About the integration variables more will be said in the next section. For the moment, note that the `X` in `XMREM5` refers to the ratio, i.e. `XMREM5` runs from 0 to 1. The actual integration borders are determined internally by `Split`.

### 5.2.3 Variables

The kinematic input variables are organized in a homogeneous system. Each variable is referred to by a preprocessor constant, e.g. `SQRTS` or `XCOSTH` (variables starting with `X` are generally scaled, i.e. run from 0 to 1). The following parts can be accessed via preprocessor macros:

- `Var(i)` = the actual value of  $i$ .
- `Show(i)` = the value printed in the output – to print e.g.  $t$  instead of  $\cos \theta$ .
- `Lower(i)`, `Upper(i)`, `Step(i)` = the lower limit, upper limit, and step width of  $i$ .

If the step is zero, the cross-section is integrated over  $i$ .

If the step is  $-999$ , the variable is considered ‘spurious’, i.e. used for output only, not integrated over or stepped through.

- $\text{CutMin}(i), \text{CutMax}(i)$  = the lower and upper cuts on  $i$ .

There are two special variables: **FIXED** for fixed values, i.e. no integration, and **TRIVIAL** for trivial integrations.

### 5.2.4 Cuts

The subroutine **Split** allows to place cuts on each  $M$ - and  $\cos\theta$ -integration. The  $\varphi$ -integration is not modified in the present setup.

Cuts restricting $M_i$		Cuts restricting $\cos\theta_i$	
Cut on	Key	Cut on	Key
$M_i$	CUT_MREM	$\cos\theta_i$	CUT_COSTH
$E_i$	CUT_MREM_E	$\cos\hat{\theta}_i$	CUT_COSTHCMS
$k_i$	CUT_MREM_K	$E_i$	CUT_COSTH_E
$E_{T,i}$	CUT_MREM_ET	$k_i$	CUT_COSTH_K
$k_{T,i}$	CUT_MREM_KT		
$y_i$	CUT_MREM_RAP		
$\eta_i$	CUT_MREM_PRAP		

In practice, the application of cuts works as follows:

```

key = 0

CutMin(XMREM5) = E5MIN
key = key + Cut(CUT_MREM_E, CUT_MIN)

CutMin(XCOSTH5) = -(1 - COSTH5CUT)
CutMax(XCOSTH5) = +(1 - COSTH5CUT)
key = key + Cut(CUT_COSTH, CUT_MIN + CUT_MAX)

call Split(5, dble(MASS5),
&  p, ex,ey,ez, minv, msum, fac, key,
&  Var(XMREM5), Var(XCOSTH5), Var(TRIVIAL))
...

```



The value of the cut is deposited in `CutMin` or `CutMax` and the cut registered by adding an identifier for the cut to the integer key, e.g. `Cut(CUT_MREM_E, CUT_MIN)` specifies a cut on the energy (`CUT_MREM_E`) from below (`CUT_MIN`) which is used to restrict the invariant-mass integration (`CUT_MREM_E`).

Note that these cuts really restrict the integration limits. They do not introduce veto functions (1 in wanted, 0 in unwanted areas) into the integrand, which can severely hamper convergence.

In addition, for each external particle  $i$  several kinematical quantities are available:

- `momspec(SPEC_M,i)` – mass  $m$ ,
- `momspec(SPEC_E,i)` – energy  $E$ ,
- `momspec(SPEC_K,i)` – momentum  $k$ ,
- `momspec(SPEC_ET,i)` – transverse energy  $E_T$ ,
- `momspec(SPEC_KT,i)` – transverse momentum  $k_T$ ,
- `momspec(SPEC_RAP,i)` – rapidity  $y$ ,
- `momspec(SPEC_PRAP,i)` – pseudo-rapidity  $\eta$ ,
- `momspec(SPEC_DELTAK,i)` – the difference  $E - k$ .

### 5.2.5 Convolution

With the new variable system, the convolution with arbitrary parton distribution functions can easily be achieved. Three modules are already included in *FormCalc* 5:

- `lumi_parton.F` = initial-state partons, no convolution.
- `lumi_hadron.F` = initial-state hadrons, convolution with hadronic PDFs from the LHAPDF library [WhBG05].
- `lumi_photon.F` = initial-state photons, convolution with CompAZ spectrum [Za03].

### 5.2.6 Integration parameters

Depending on the integrand, the actual integration can be fairly tricky to carry out numerically. `2to3.F` employs the CUBA library [Ha04] which offers four integration routines. The CUBA parameters are chosen in `run.F` as preprocessor variables:

```

#define METHOD DIVONNE
#define RELACCURACY 1D-3
#define ABSACCURACY 1D-7
#define VERBOSE 1
#define MINEVAL 0
#define MAXEVAL 50000

* for Vegas:
#define NSTART 1000
#define NINCREASE 500

* for Suave:
#define NNEW 1000
#define FLATNESS 50

* for Divonne:
#define KEY1 47
#define KEY2 1
#define KEY3 1
#define MAXPASS 5
#define BORDER 1D-6
#define MAXCHISQ 10
#define MINDEVIATION .25D0

* for Cuhre:
#define KEY 0

```

The integration algorithm is selected with `METHOD`, which can take the values `VEGAS`, `SUAVE`, `DIVONNE`, and `CUHRE`. The other preprocessor variables determine parameters of the integrators and may/should be tuned for a particular integrand, for details see [Ha04].

### 5.2.7 Compiling and running the code

The code produced by `WriteSquaredME` is compiled with the commands

```

./configure
make

```

The `configure` script searches for the compilers and necessary libraries and writes out a `makefile` by adding the appropriate locations and flags to `makefile.in`. The ‘usual’ envi-

environment variables like FC (Fortran compiler) and FFLAGS (Fortran flags) can be used to force particular choices.

Based on the makefile, the make command then builds the executable. Its default name is run, which is quite natural because the Fortran compiler sees run.F as the main program, as mentioned before. The advantage is that for a different run, one can make a copy of run.F, say run1.F, with different parameters. This new run is compiled with “make run1” and results in the executable run1. Thus, one can have several executables for different runs in one directory.

The way the makefile compiles the code is also convenient if one wants to use the generated subroutine SquaredME alone, i.e. without the surrounding driver programs. The necessary object files are all placed in the library squaredme.a so that only a single file needs to be linked. It is possible to build just this library with

```
make squaredme.a
```

The executables (run, run1, etc.) are able to calculate differential and integrated cross-sections for particles of arbitrary polarization depending on the command line.

<code>run <math>p_1 \dots p_n</math> sqrtS [serialfrom[,serialto[,serialstep]]]</code>	compute the differential cross-section at a centre-of-mass energy <i>sqrtS</i> with the external particles polarized according to $p_1 \dots p_n$ .
<code>run <math>p_1 \dots p_n</math> sqrtSfrom,sqrtSto[,sqrtSstep] [serialfrom[,serialto[,serialstep]]]</code>	compute the integrated cross-section in the energy range <i>sqrtSfrom</i> – <i>sqrtSto</i> in steps of <i>sqrtSstep</i> , $p_1 \dots p_n$ being the polarizations of the external particles.

The  $p_i$  can take the following values:

- u for an unpolarized particle,
- t for a transversely polarized particle,
- + for right-circular polarization,
- for left-circular polarization, and
- 0 for longitudinal polarization.

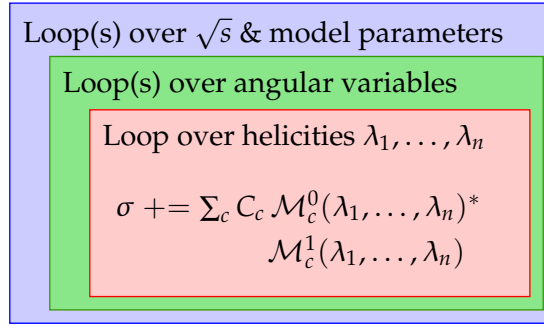
For a scalar particle, the polarization parameter must be present on the command line although its value is ignored. The energies are specified in GeV. If an energy is below threshold, run issues a warning and sets the energy to the threshold + 0.01 GeV.

A serial number range may optionally be added as a third argument, restricting the parameter scans. This is normally used only internally by the submit script (see below).

*Important:* If a particular polarization was already fixed during the algebraic simplification, e.g. if `_Hel = 0` was set in *FormCalc* to calculate with unpolarized external fermions, the code thus produced can never compute a cross-section for a different polarization, no matter how the executable is invoked. Running such code with an incompatible polarization will in general only result in a wrong averaging factor.

### 5.2.8 Parallelization and Vectorization

The assembly of the squared matrix element in code generated by *FormCalc* can be sketched as in the following figure, where the helicity loop sits at the center of the calculation:



The helicity loop is not only strategically the most desirable but also the most obvious candidate for concurrent execution. *FormCalc* can distribute the helicity loop in two ways: parallel execution via `fork/wait` on the cores of the local CPU (i.e. SMP), and vectorization using the CPU's SIMD instructions (that is, the squared matrix element is computed for several helicity combinations at once), or both. The include file `distrib.h` chooses the distribution properties of the compiled code. Distribution of scans over parameter space is described in Sect. 5.2.9.

Parallel execution with `fork/wait` is switched on by setting the preprocessor variable `PARALLEL` in `distrib.h` to 1. The drawback of this method is that it competes for compute cores in particular with *Cuba*. In the usual settings of few (e.g. 4 or 8) cores per system the cores should be assigned to *Cuba* since it computes entire phase-space points in parallel, not just the helicity loop. The number of cores is determined from the environment variable `FCCORES`, or if that is not set, from the number of cores reported by the system minus the current load average.

Vectorization is turned on automatically by the `configure` script if it detects that the CPU and compiler are capable of it. If this is in conflict with plans to send the executable on a cluster of computers with heterogeneous SIMD capabilities, run `configure` with the `--generic` option. Then again, SIMD instructions are typically available on recent computers: SSE3 since 2004, AVX since 2011. To control vectorization 'by hand' one needs to specify the preprocessor variable `SIMD` in `distrib.h` rather than including the file `simd.h` generated by `configure`.

SIMD specifies the length of the vector, sensible values are 2 for AVX, 1 for SSE3, and 0 otherwise. In Fortran the vector length can in principle been chosen arbitrarily though a speedup is expected only for the mentioned values commensurate with the hardware.

On top of this, the calculation automatically detects and omits helicity combinations contributing negligibly. This feature is controlled by the two environment variables FCHSELN and FCHSELEPS in the following way: for the first FCHSELN phase-space points, the computation runs over all helicity combinations  $h$  and the absolute value of the squared matrix element is added up in an array  $t(h)$ . For subsequent phase-space points, then, only helicity combinations with a  $t$ -value larger than  $(\text{FCHSELEPS} \cdot \max_h t(h))$  are actually computed. Unless one intentionally begins sampling e.g. on the borders of phase-space, the default values  $\text{FCHSELN} = 10$  and  $\text{FCHSELEPS} = 10^{-7}$  should be sufficient. In case of doubt about the validity of the result, the value of FCHSELN should be increased, or set to zero (which turns the detection off).

If algebraic relations between helicities are known from analytic considerations, these can be specified in Mathematica before calling WriteSquaredME and will restrict the sum over helicities accordingly. For example:

```
Hel[2] = -Hel[1];
WriteSquaredME[...]
```

### 5.2.9 Scans over parameter space

To perform a scan, the actual calculation needs to be enclosed in a number of do-loops in which the values of the scanned parameters are changed, i.e. a structure of the form

```
do 1 para1 = ...
do 1 para2 = ... etc.
  calculate the cross-section
1  continue
```

run.F provides several preprocessor variables, LOOP1...LOOP15, which may be defined to contain the do-statements that initiate the loops. Each loop must terminate on statement 1, just as above. For example, the lines

```
#define LOOP1 do 1 MT = 160, 180, 5
#define LOOP2 do 1 MH = 100, 300, 100
```

result in a scan of the form

```
do 1 MT = 160, 180, 5
do 1 MH = 100, 300, 100
```

```

        calculate the cross-section
1      continue

```

The loops are nested inside each other with LOOP1 being the outermost loop. It is not mandatory to declare a loop with each LOOP $n$ , also a fixed value may be given as in the following definition:

```
#define LOOP1 MH = 115
```

When scanning over parameter space, it is further necessary to keep track of the parameter values for each data set. This is done by defining PRINT commands in the preprocessor variables PRINT1...PRINT5 in run.F. The PRINT command takes pairs of arguments, first a string identifying the variable and then the variable itself. Defining, for example,

```
#define PRINT1 PRINT "MT=", MT, "MH=", MH
```

causes each data set to be preceded by two lines of the form

```

# MT= 160.00
# MH= 100.00

```

Such a scan can be a real CPU hog, but on the other hand, the calculation can be performed completely independently for each parameter set and is thus an ideal candidate for parallelization. The real question is thus not how to parallelize the calculation, but how to automate the parallelization.

The obstacle to automatic parallelization is that the loops are user-defined and in general nested. A serial number is introduced to unroll the loops:

```

    serial = 0
    LOOP1
    LOOP2
    :
    serial = serial + 1
    if( serial not in allowed range ) goto 1
    calculate cross-section
1  continue

```

As mentioned before, the serial number range can be specified on the command line so that it is quite straightforward to distribute patches of serial numbers on different machines. Most easily this is done in an interleaved manner, since one then does not need to know to which upper limit the serial number runs, i.e. if there are  $N$  machines available, send serial numbers

1,  $N + 1$ ,  $2N + 1$ , ... on machine 1, send serial numbers 2,  $N + 2$ ,  $2N + 2$ , ... on machine 2, etc.

This procedure is automated in *FormCalc*: The user once creates a `.submitrc` file in his home directory and lists there all machines that may be used, one on each line. The only requirement is that the machines are binary compatible because the same executable will be started on each. In the case of multi-processor machines the number of processors is given after the host name. Empty lines and lines beginning with `#` are treated as comments. Furthermore, a line of the form `nice n` determines the nice value at which the remote jobs are started. For example:

```
# .submitrc for FormCalc for the institute cluster
# start jobs with nice 10:
nice 10

pcxeon1 8
pcxeon2 8

pcjoe
pcath6
pcath7
```

The executable compiled from *FormCalc* code, typically called `run`, is then simply prefixed with `submit`. For instance, instead of

```
run uuuu 500,1000
```

the user invokes

```
submit run uuuu 500,1000
```

The `submit` script uses `uptime` to determine the load of the machines and `ssh` to log in. Handling of the serial number is invisible to the user.

### 5.2.10 Log files, Data files, and Resume

Due to the parallelization mechanism, a single output file is not sufficient. Instead, *FormCalc* creates a directory for each invocation of `run`, e.g. `run.UUUU.00200` for a differential cross-section or `run.UUUU.00200-00500:00010` for an integrated cross-section (where 00010 is the step width in the loop over the energy), opens one log file for each serial number in this directory, and redirects console output to this file.

Each log file contains both the ‘real’ data and the ‘chatter’ (progress, warning, and error messages). This has the advantage that no unit numbers must be passed between subroutines – every bit of output is simply written to the console (unit \* in Fortran). It also makes it easier to pinpoint errors, since the error message appears right next to the corrupted data. The ‘real’ data are marked by a “|” in column 1 and there exists a simple shell script, `data`, to extract the real data from the log file. For example,

```
data run.UUUU.00200
```

creates a data file `run.UUUU.00200.data` containing only the ‘real’ data arranged in three columns, where the first column is the scattering angle in radians, the second column is the tree-level cross-section, and the third column is the one-loop correction. For an integrated cross-section, there are five columns:  $\sqrt{s}$ , the tree-level cross-section, the one-loop correction, and the integration errors for the tree-level and one-loop cross-sections. Cross-sections are computed in picobarn.

The log-file management also provides an easy way to resume an aborted calculation. This works as follows: when running through the loops of a parameter scan, the log file for a particular serial number

- may not exist: then it is created with execute permissions,
- may exist, but have execute permissions: then it is overwritten,
- may exist and have read-write permissions: then this serial number is skipped.

The execute permissions, which serve here merely as a flag to indicate an ongoing calculation, are reduced to ordinary read-write permissions when the log file is closed.

In other words, the program skips over the parts of the calculation that are already finished, so all the user has to do to resume an aborted calculation is start the program again with the same parameters.

### 5.2.11 Shell scripts

`turnoff` switches off (and on) the evaluation of certain parts of the amplitude, which is a handy thing for testing. For example, “`./turnoff box`” switches off all parts of the amplitude with ‘box’ in their name. Invoking `turnoff` without any argument restores all modules.

`sfx` packs all source files (but not object, executable, or log files) in the directory it is invoked in into a mail-safe self-extracting archive. For example, if `sfx` is invoked in the directory `myprocess`, it produces `myprocess.sfx`. This file can e.g. be mailed to a collaborator, who needs to say “`./myprocess.sfx x`” to unpack the contents.



`pnuglot` produces a high-quality plot in Encapsulated PostScript format from a data file in just one line. In fact, `pnuglot` does not even make the plot itself, it writes out a shell script to do that, thus “`./pnuglot mydata`” creates `mydata.gpl` which then runs `gnuplot`,  $\text{\LaTeX}$ , and `dvips` to create `mydata.eps`. The advantage of this indirect method is that the default `gnuplot` commands in `mydata.gpl` can subsequently be edited to suit the user’s taste. Adding a label or choosing a different line width is, for example, a pretty trivial matter. Needless to say, all labels are in  $\text{\LaTeX}$  and Type 1 fonts are selected to make the EPS file nicely scalable.

`pnuglot` by default uses commands only available in `gnuplot` version 3.7 or higher. This version can be obtained from <http://www.gnuplot.info>.

<code>pnuglot</code>	<code>[opts]</code>	<code>file<sub>1</sub> file<sub>2</sub> ...</code>	make a plot of the data files <i>file<sub>1</sub></i> , <i>file<sub>2</sub></i> , ...
		<i>options:</i>	
	<code>-o</code>	<i>outfile</i>	how to name the output file: the plotting script will be called <i>outfile.gpl</i> and the actual plot <i>outfile.eps</i> . The default is <i>outfile = file<sub>1</sub></i> .
	<code>-2</code>		use only columns 1:2 (usually the tree-level cross-section) for plotting
	<code>-3</code>		use only columns 1:(2+3) (usually the one-loop corrected cross-section) for plotting

### 5.3 The Mathematica Interface

The Mathematica interface turns the stand-alone code into a Mathematica function for evaluating the cross-section or decay rate as a function of user-selected model parameters. The benefits of such a function are obvious, as the whole instrumentarium of Mathematica commands can be applied to them. For example, it is quite straightforward, using Mathematica’s `FindMinimum`, to determine the minimum (or maximum) of the cross-section over a piece of parameter space.

Interfacing is done using the MathLink protocol. The changes necessary to produce a MathLink executable are quite superficial and affect only the file `run.F`, where the user has to choose which model parameters are interfaced from Mathematica.

To make the obvious even clearer, the cross-section is *not* evaluated in Mathematica, but in Fortran or C, and only the numerical results are transferred back to Mathematica. One thing one cannot do thus is to increase the numerical precision of the calculation using Mathematica commands like `SetPrecision`.

### 5.3.1 Setting up the Interface

The model parameters are specified in the file `run.F`. Typical definitions for stand-alone code look like (here from an MSSM calculation with  $TB = \tan \beta$  and  $MA0 = M_{A0}$ ):

```
#define LOOP1 do 1 TB = 5, 50, 5
#define LOOP2 MA0 = 500
...
```

These lines declare  $TB$  to be scanned from 5 to 50 in steps of 5 and set  $MA0$  to 500 GeV. To be able to specify  $TB$  in Mathematica instead, the only change is

```
#define LOOP1 call MmaGetReal(TB)
```

Such invocations of `MmaGetReal` and its companion subroutines serve two purposes. At compile time they determine with which arguments the Mathematica function is generated (for details see below), and at run time they actually transfer the function's arguments to the specified variables.

<code>MmaGetInteger(<i>i</i>)</code>	read the integer/real/complex parameter <i>i/r/c</i>
<code>MmaGetReal(<i>r</i>)</code>	from Mathematica
<code>MmaGetComplex(<i>c</i>)</code>	
<code>MmaGetIntegerList(<i>i</i>, <i>n</i>)</code>	read the integer/real/complex parameter list <i>i/r/c</i>
<code>MmaGetRealList(<i>r</i>, <i>n</i>)</code>	of length <i>n</i> from Mathematica
<code>MmaGetComplexList(<i>c</i>, <i>n</i>)</code>	

Note that without a separate `MmaGetReal` call,  $MA0$  would still be fixed by the Fortran statement, i.e. not be accessible from Mathematica.

Once the makefile detects the presence of these subroutines, it automatically generates interfacing code and compiles a MathLink executable. For a file `run.F` the corresponding MathLink executable is also called `run`, as in the stand-alone case. This file is not started from the command-line, but used in Mathematica as

```
Install["run"]
```

### 5.3.2 The Interface Function in Mathematica

After loading the MathLink executable with `Install`, a Mathematica function of the same name is available. For definiteness, we will call this function 'run' in the following since 'run.F' is the default parameter file. This function has the arguments

```
run[sqrtS, arg1, arg2, ..., options]
```

```
run[{sqrtSfrom, sqrtSto[, sqrtSstep]}, arg1, arg2, ..., options]
```

The first form computes a differential cross-section at  $\sqrt{s} = \text{sqrtS}$ . The second form computes a total cross-section for energies  $\sqrt{s}$  varying from  $\text{sqrtSfrom}$  to  $\text{sqrtSto}$  in steps of  $\text{sqrtSstep}$ . This is in one-to-one correspondence with the command-line invocation of the stand-alone executable.

The  $\text{arg}_1, \text{arg}_2, \dots$ , are the model parameters declared automatically by the presence of their `MmaGet{Real,Complex}` calls (see above). They appear in the argument list in the same order as the corresponding `MmaGet{Real,Complex}` calls.

Other parameters are specified through the *options*.

Default Value	default value	
Polarizations	"UUUU"	the polarizations of the external particles
Serial	{}	the range of serial numbers to compute
SetNumber	1	a set number beginning with which parameters and data are stored
ParaHead	Para	the head under which parameters are stored
DataHead	Data	the head for the data storage
LogFile	" "	the log file to save screen output in

Polarizations determines the polarizations of the external particles, specified as in the stand-alone version, i.e. a string of characters for each external leg:

U	unpolarized,	L	longitudinal polarization,
T	transversely polarized,	-	left-handed polarization,
		+	right-handed polarization.

Serial gives the range of serial numbers for which to perform the calculation, specified as `{serialfrom[, serialto[, serialstep]]}`. The concept of serial numbers, used to parallelize parameter scans, is described in Sect. 5.2.9. This option applies only to parameters scanned by do-loops in the parameter statements. Parameters read from Mathematica are unaffected by this option.

SetNumber specifies the set number beginning with which parameters and data are stored (see next Section).

ParaHead gives the head under which parameters are stored, i.e. parameters are retrievable from `parahead[setnumber]` (see next Section).

DataHead gives the head under which data are stored, i.e. data are retrievable from `datahead[setnumber]` (see next Section).

LogFile specifies the log-file to save screen output in. An empty string indicates no output redirection, i.e. the output will appear on screen.

### 5.3.3 Return values, Storage of Data

The return value of the generated function is an integer which records how many parameter and data sets were transferred. Assigning parameter and data sets as the data become available has several advantages:

- the return value of `run` is an integer rather than a large, bulky list,
- the parameters corresponding to a particular data set are easy to identify, e.g. `Para[4711]` contains the parameters corresponding to `Data[4711]`,
- most importantly, if the calculation is prematurely aborted, the parameters and data transferred so far are still accessible.

Both, the starting set number and the heads of the parameter and data assignments can be chosen with the options `SetNumber`, `ParaHead`, and `DataHead`, respectively.

The parameters which are actually returned are chosen by the user in the `PRINT $n$`  statements in `run.F` in much the same way as parameters are selected for printout in the stand-alone code. To specify that `TB` and `MA0` be returned, one needs the definitions

```
#define PRINT1 call MmaPutReal("TB", TB)
#define PRINT2 call MmaPutReal("MA0", MA0)
```

Notwithstanding, parameters can still be printed out, in which case they end up in the log file (or on screen, if no log file is chosen). To transfer e.g. `TB` to Mathematica *and* print it out, one would use

```
#define PRINT1 call MmaPutReal("TB", TB)
#define PRINT2 SHOW "TB", TB
```

An analogous subroutine exists of course for integer and complex parameters, too.

<code>MmaPutInteger(s, i)</code>	transfer the integer/real/complex parameter $i/r/c$
<code>MmaPutReal(s, r)</code>	to Mathematica under the name $s$
<code>MmaPutComplex(s, c)</code>	
<code>MmaPutIntegerList(s, i, n)</code>	transfer the integer/real/complex parameter list
<code>MmaPutRealList(s, r, n)</code>	$i/r/c$ of length $n$ to Mathematica under the name $s$
<code>MmaPutComplexList(s, c, n)</code>	

The parameters are stored in the form of rules in Mathematica, i.e. as *name*  $\rightarrow$  *value*. The first argument specifies the left-hand side of this rule. It need not be a symbol in the strict sense, but can be an arbitrary Mathematica expression. But note that in particular the underscore has a special meaning in Mathematica and may not be used in symbol names. The second argument is then the right-hand side of the rule and can be an arbitrary Fortran expression containing model parameters, kinematic variables, etc.

The following example demonstrates the form of the parameter and data assignments. Shown are results of a differential cross-section for a  $2 \rightarrow 2$  reaction at one point in MSSM parameter space. Within the data the varied parameter is  $\cos \theta$ , the scattering angle.

```
Para[1] = { TB -> 1.5, MUE -> -1000., MSusy -> 1000.,
           MA0 -> 700., M2 -> 100. }

Data[1] = { DataRow[{500., -0.99},
                    {0.10592302458950732, 0.016577997941111422},
                    {0., 0.}],
           DataRow[{500., -0.33},
                    {0.16495552191438356, 0.014989931149150608},
                    {0., 0.}],
           DataRow[{500., 0.33},
                    {0.2986891221231292, 0.015013326141014818},
                    {0., 0.}],
           DataRow[{500., 0.99},
                    {0.5071238252157443, 0.012260927614082411},
                    {0., 0.}] }
```

<code>DataRow[v, r, e]</code>	a row of data with kinematic variables $v$ , cross-section or decay-rate results $r$ , and integration error $e$
-------------------------------	--

The `DataRow[v, r, e]` function has three arguments:

- the unintegrated kinematic variables ( $v = \{\sqrt{s}, \cos \theta\}$  above),

- the cross-section or decay-rate results ( $r = \{\text{tree-level result, one-loop correction}\}$  above), and
- the respective integration errors ( $e = \{0, 0\}$  above, as this example originates from the computation of a differential cross-section where no integration is performed).

### 5.3.4 Using the Generated Mathematica Function

To the Mathematica novice it may not be obvious how to use the function described above to analyse data, produce plots, etc.

As an example, let us produce a contour plot of the cross-section in the  $M_{A^0}$ - $\tan \beta$  plane. It is assumed that the function run has the two parameters MA0 and TB in its argument list:

```
Install["run"]

xs[sqrtS_, MA0_, TB_] := (
  run[{sqrtS, sqrtS}, MA0, TB];
  Data[1][[1, 2]] )

ContourPlot[xs[500, MA0, TB], {MA0, 100, 500}, {TB, 5, 50}]
```

The function `xs` runs the code and selects the data to plot. The first argument of `run`, `{sqrtS, sqrtS}`, instructs the code to compute the total cross-section for just one point in energy. We then select the first (and only) `DataRow` in the output and choose its second argument, the cross-section results: `Data[1][[1, 2]]`.

This example can be extended a little to produce a one-dimensional plot where e.g. for each value of  $\tan \beta$  the minimum and maximum of the cross-section with respect to  $M_{A^0}$  is recorded:

```
<< Graphics`FilledPlot`

xsmin[sqrtS_, TB_] :=
  FindMinimum[xs[sqrtS, MA0, TB], {MA0, 100}][[1]]
xsmax[sqrtS_, TB_] :=
  -FindMinimum[-xs[sqrtS, MA0, TB], {MA0, 100}][[1]]

FilledPlot[{xsmin[500, TB], xsmax[500, TB]}, {TB, 5, 50}]
```

## 5.4 Renormalization Constants

*FormCalc* provides a number of functions to facilitate the computation of renormalization constants (RCs). *FormCalc* makes a conceptual distinction between the definition and the calculation of RCs.

### 5.4.1 Definition of renormalization constants

*FormCalc* regards those symbols as RCs for which a definition of the form

```
RenConst[ rc ] := ...
```

exists. The purpose of the definition is to provide the functional relationship between the RC and the self-energy from which it is calculated, for example

```
RenConst[ dMHsq1 ] := ReTilde[SelfEnergy[S[1] -> S[1], MH]]
```

Note that this definition is quite generic, in particular it contains no details about the selection of diagrams. It is intentional that the specifics of the diagram generation and calculation are chosen only when the RCs are actually calculated because this allows the user to make selections on a process-by-process basis. For example, one can calculate the main process and the RCs in one program such that the options chosen for the former (e.g. for `InsertFields`) automatically apply to the latter.

The definitions of the RCs can be made anywhere before one of the functions that calculates them is invoked. A particularly convenient location is in the model file to which the RCs belong. This is the case for the model files that come with the current versions of *FeynArts*, where the definitions of the RCs are implemented according to the on-shell scheme of [De93]. The definition of an RC may make use of the following functions. They may be used in the model file or anywhere else, even if *FormCalc* is not loaded.

<code>SelfEnergy[i -&gt; f, m]</code>	the self-energy $\Sigma_{if}(k^2 = m^2)$
<code>DSelfEnergy[i -&gt; f, m]</code>	the derivative $\partial \Sigma_{if}(k^2) / \partial k^2 \big _{k^2=m^2}$
<code>TreeCoupling[i -&gt; f]</code>	the tree-level contribution to $i \rightarrow f$ without external spinors
<code>LVectorCoeff[expr]</code>	the coefficient of $\not{k}\omega_-$ in <i>expr</i>
<code>RVectorCoeff[expr]</code>	the coefficient of $\not{k}\omega_+$ in <i>expr</i>
<code>LScalarCoeff[expr]</code>	the coefficient of $\omega_-$ in <i>expr</i>
<code>RScalarCoeff[expr]</code>	the coefficient of $\omega_+$ in <i>expr</i>
<code>ReTilde[expr]</code>	takes the real part of loop integrals in <i>expr</i>
<code>ImTilde[expr]</code>	takes the imaginary part of loop integrals in <i>expr</i>

Most RCs fall into one of the following categories, for which special functions exist to save typing and reduce errors. (The width is of course not an RC, but is also listed here as its computation is very similar to that of an RC.)

<code>MassRC[f]</code>	the mass RC $\delta M_f$
<code>MassRC[f<sub>1</sub>, f<sub>2</sub>]</code>	the mass RC $\delta M_{f_1 f_2}$
<code>FieldRC[f]</code>	the field RC $\delta Z_f$
<code>FieldRC[f<sub>1</sub>, f<sub>2</sub>]</code>	the field RC $\delta Z_{f_1 f_2}$
<code>TadpoleRC[f]</code>	the tadpole RC $\delta T_f$
<code>WidthRC[f]</code>	the width $\Gamma_f$

The explicit formulas for computing the RCs are given in the following. For compactness of notation,  $f$  refers to a fermion and  $B$  to a boson field and  $\partial\Sigma(m^2)$  is short for  $\partial\Sigma(k^2)/\partial k^2|_{k^2=m^2}$ .

$$\begin{aligned}
\delta M_f &= \widetilde{\text{Re}} \Sigma_{ff}^F(m_f, \tfrac{1}{2}, \tfrac{1}{2}), & \delta M_B &= \widetilde{\text{Re}} \Sigma_{BB}(m_B^2), \\
\delta M_{B_1 B_2} &= \frac{1}{2} \widetilde{\text{Re}} \left( \Sigma_{B_2 B_1}(m_{B_1}^2) + \Sigma_{B_2 B_1}(m_{B_2}^2) \right), \\
\delta Z_f &= -\widetilde{\text{Re}} \left[ \begin{array}{c} \Sigma_{ff}^{VL} + \partial\Sigma_{ff}^F(m_f, m_f, m_f) \\ \Sigma_{ff}^{VR} + \partial\Sigma_{ff}^F(m_f, m_f, m_f) \end{array} \right], & \delta Z_B &= -\widetilde{\text{Re}} \partial\Sigma_{BB}(m_B^2), \\
\delta Z_{f_1 f_2} &= \frac{2}{m_{f_1}^2 - m_{f_2}^2} \widetilde{\text{Re}} \left[ \begin{array}{c} \Sigma_{f_2 f_1}^F(m_{f_2}, m_{f_2}, m_{f_1}) \\ \Sigma_{f_2 f_1}^F(m_{f_2}, m_{f_1}, m_{f_2}) \end{array} \right], & \delta Z_{B_1 B_2} &= \frac{2}{m_{B_1}^2 - m_{B_2}^2} \widetilde{\text{Re}} \Sigma_{B_2 B_1}(m_{B_2}^2), \\
\delta T_B &= -\Sigma_B(m_B^2), \\
\Gamma_f &= \widetilde{\text{Im}} \Sigma_{ff}^F(m_f, 1, 1), & \Gamma_B &= \frac{1}{m_B} \widetilde{\text{Im}} \Sigma_{BB}(m_B^2),
\end{aligned}$$

where

$$\Sigma^F(m, \alpha, \beta) = m [\alpha \Sigma^{VL}(m^2) + \beta \Sigma^{VR}(m^2)] + \beta \Sigma^{SL}(m^2) + \alpha \Sigma^{SR}(m^2).$$

#### 5.4.2 Calculation of renormalization constants

For the actual calculation of the RCs, *FormCalc* provides the two functions `CalcRenConst` and `WriteRenConst`. Both functions search the expressions they receive as arguments for symbols which are RCs. `CalcRenConst` returns the calculated RCs as a list of rules whereas `WriteRenConst` writes them to a program file.



<code>CalcRenConst[expr]</code>	calculate the RCs appearing in <i>expr</i> and return the results as a list of rules
<code>WriteRenConst[expr, dir]</code>	the same, but write the results to a program in the code directory <i>dir</i>

`WriteRenConst` writes out the subroutine `CalcRenConst` as well as the declarations of the RCs in `renconst.h`. It shares a number of options regarding the technicalities of code generation with `WriteSquaredME`. In the following table, if the default value coincides with the option name this means that the value is taken from the `WriteSquaredME` options.

<i>option</i>	<i>default value</i>	
<code>Folder</code>	<code>"renconst"</code>	the subdirectory of the code directory into which the generated code is written
<code>FilePrefix</code>	<code>FilePrefix</code>	a string prepended to the filenames of the generated code
<code>SymbolPrefix</code>	<code>SymbolPrefix</code>	a string prepended to global symbols to prevent collision of names when more than one process is linked
<code>FileHeader</code>	<code>FileHeader</code>	the file header
<code>FileIncludes</code>	<code>FileIncludes</code>	per-file <code>#include</code> statements
<code>SubroutineIncludes</code>	<code>Subroutine\Includes</code>	per-subroutine <code>#include</code> statements

At this point it is necessary to discuss how selections of diagrams can be made for the self-energies. The functions `SelfEnergy` and `DSelfEnergy`, which are used in the definitions of the RCs, are invoked when the RCs are calculated. They generally use what options of `InsertFields`, `CreateFeynAmp`, and `CalcFeynAmp` are in place at that time. Additionally, they provide a hook for the user to customize the exact use of `InsertFields`. That is, `SelfEnergy` and `DSelfEnergy` do not call `InsertFields` directly, but a function called `InsertFieldsHook` whose default definition is

```
InsertFieldsHook[tops_, proc_] := InsertFields[tops, proc]
```

which simply inserts fields into the topologies `tops` for the process `proc`. The function `InsertFieldsHook` can be redefined either generally or for a specific process, e.g.

```
InsertFieldsHook[tops_, f1_F -> f2_F] :=  
  InsertFields[tops, f1 -> f2, ExcludeParticles -> V[1]]
```

would exclude photons in fermion self-energies. One could also imagine more complicated

definitions, e.g. involving `DiagramSelect`.

Furthermore, it is possible to have the diagrams painted by setting the variable `$PaintSE`. `$PaintSE` can be `True`, `False`, or a string which indicates that the output should be saved in a PostScript file instead of being displayed on screen. If a string is given, it is used as a prefix for the name of the PostScript file, hence it is possible to specify e.g. a path.

Finally, it is possible to specify options for individual RCs, as in

```
Options[ dMWsq1 ] = {Restrictions -> NoSUSYParticles}
```

These options are taken as `InsertFields` options which apply only to the calculation of this particular renormalization constant.

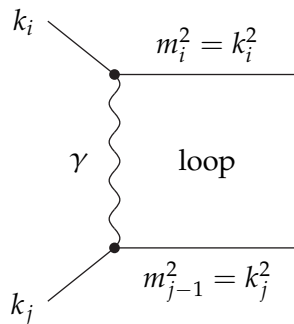
For performance reasons, `SelfEnergy` and `DSelfEnergy` remember the values they have computed. In unlikely cases it may happen that the user changes settings that affect the generation of diagrams but are not recognized by `SelfEnergy` and `DSelfEnergy`. This would mean that results are returned which are no longer correct. In such a case, the cached values can be removed with `ClearSE[]`.

Similar to the functions `FormSub`, `FormDot`, etc., there are simplification wrappers specifically for RCs:

<code>RCSub[subexpr]</code>	a function applied to subexpressions extracted by FORM
<code>RCInt[intcoeff]</code>	a function applied to the coefficients of loop integrals in the FORM output

## 5.5 Infrared Divergences and the Soft-photon Factor

Infrared divergences appear in processes with charged external particles. They originate from the exchange of virtual photons. More precisely they come from diagrams containing structures of the form



Such diagrams are IR divergent because the photon is massless; if the photon had a mass  $\lambda$ , the divergent terms would be proportional to  $\log \lambda$ .

A note on the numerical implementation: In *LoopTools* it is not necessary to introduce a photon mass by hand: if a requested integral is IR divergent, *LoopTools* automatically regularizes the divergence with a photon mass  $\lambda$ , but treats  $\lambda$  as an infinitesimal quantity, which means that terms of order  $\lambda$  or higher are discarded. In practice,  $\lambda$  is a user-definable numerical constant. Since the final result should not depend on it after successful removal of the IR divergences, it can be given an arbitrary numerical value despite its infinitesimal character.

The divergences in the diagrams with virtual photon exchange exactly cancel those in the emission amplitude of soft (i.e. low-energetic) photons off the external legs. Soft-photon radiation is always present experimentally: because photons are massless, they can have arbitrarily low energies and therefore escape undetected. So while the unphysical  $\lambda$ -dependence cancels, the final result depends instead on a detector-dependent quantity  $E_{\text{soft}}^{\text{max}}$ , the maximum energy a soft photon may have without being detected.

In a one-loop calculation one needs to take into account the bremsstrahlung off the Born diagrams to be consistent in the order of  $\alpha$ . Dealing with soft photons is relatively easy since the soft-photon cross-section is always proportional to the Born cross-section:

$$\left[ \frac{d\sigma}{d\Omega} \right]_{\text{SB}} = \delta_{\text{SB}} \left[ \frac{d\sigma}{d\Omega} \right]_{\text{Born}}$$

where “SB” stands for soft bremsstrahlung.

The soft-photon factor  $\delta_{\text{SB}}$  is implemented in `main.F` using the formulas of [De93]. It is controlled by the following two preprocessor statements in `process.h`.

```
c#define BREMSSTRAHLUNG
#define ESFTMAX .1D0*sqrtS
```

The first flag, `BREMSSTRAHLUNG`, is commented out by default. To switch on soft-photon corrections, take out the `c` before the `#define`. The second line determines the energy cutoff  $E_{\text{soft}}^{\text{max}}$ .

## 6 Post-processing of the Results

### 6.1 Reading the data files into *Mathematica*

The format in which data are written out by `main.F` is directly practical only for making plots of the cross-section over the energy or some phase-space variable. In many circumstances one needs to make more sophisticated figures in which the data are either arranged differently, or are run through some filter or calculation. Consider, for example, performing

a scan over some region in parameter space and then wanting to plot the band of allowed values of the cross-section, i.e. the minimum and maximum with respect to the parameter variations for each point in phase space.

Given the variety of operations one could conceivably perform on the data, a Fortran or C program would sooner or later prove too inflexible. Instead, there is a utility program with which it is possible to read the data into *Mathematica*, where it is much easier to further process the data sets and eventually produce plots of whatever function of whatever data set one is interested in.

The data files are read into *Mathematica* with the *MathLink* program `ReadData`, which is used in the following way:

```
In[1]:= Install["ReadData"];

In[2]:= ReadData["run.UUUU.00300-01000.data"]

Out[2]= 24
```

The data are stored in *Mathematica* as separate data and parameter sets. These sets are accessed through an integer argument, e.g. `Data[1]` and `Para[1]`. `ReadData` returns the number of sets read (24 in this example).

The data files have to be roughly in the format used by `main.F`, i.e. a number of lines beginning by `#` in which the parameters are given, followed by columns of data. The following example shows this format:

```
# TB=      1.50
# MUE=-1000.00
# MSusy= 1000.00
# MA0=   700.00
# M_2=   100.00
    160.790000000000    0.264226254743272    0.502096473139054
    190.482475649966    10.7918866098049     9.58435238790029
    211.567524166608    11.8170727823835    10.3862900941921

# TB=      1.50
# MUE=-1000.00
# MSusy= 1000.00
# MA0=   700.00
# M_2=   400.00
    160.790000000000    0.264226254743272    0.502510502734037
```

```

190.482475649966      10.7918866098049      9.60119470492217
211.567524166608      11.8170727823835      10.4047052203442

```

The output of ReadData for this file is 2, hence there are then two data and two parameter sets defined,

```

Para[1] = { TB -> 1.5, MUE -> -1000., MSusy -> 1000.,
           MA0 -> 700., M$2 -> 100. }

```

```

Data[1] = { {160.79, 0.264226, 0.502096},
            {190.482, 10.7919, 9.58435},
            {211.568, 11.8171, 10.3863} }

```

```

Para[2] = { TB -> 1.5, MUE -> -1000., MSusy -> 1000.,
           MA0 -> 700., M$2 -> 400. }

```

```

Data[2] = { {160.79, 0.264226, 0.502511},
            {190.482, 10.7919, 9.60119},
            {211.568, 11.8171, 10.4047} }

```

Note that the numbers have not suffered in accuracy; it is a peculiarity of *Mathematica* to display real numbers with no more than six digits by default. Note also that the underscore in M\_2 which is not allowed in symbol names in *Mathematica* has been replaced by a \$.

<code>ReadData[f, n, h<sub>para</sub>, h<sub>data</sub>]</code>	read the data file <i>f</i> into parameter and data sets in <i>Mathematica</i> ; with the optional parameters <i>n</i> , <i>h<sub>para</sub></i> , and <i>h<sub>data</sub></i> one can choose to start numbering the sets with <i>n</i> , use head <i>h<sub>para</sub></i> instead of Para, and use head <i>h<sub>data</sub></i> instead of Data
<code>Data[n]</code>	the <i>n</i> th data set
<code>Para[n]</code>	the <i>n</i> th parameter set

In addition, ReadData has a handy way of applying operations to each line of data: it does not, in fact, put each line of numbers in a list, but in a function called DataRow. By default, DataRow is equal to List. Now consider that instead of the absolute values of the cross-section, which is what main.F calculates by default, you want e.g. the relative deviation of the one-loop and the tree-level cross-section. All it needs to achieve this is to redefine the DataRow function as follows:

```

Clear[DataRow];
DataRow[x_, tree_, loop_] := {x, loop/tree}

```

(This works because `main.F` puts the tree-level result in the second column and the one-loop correction in the third column.)

<i>function</i>	<i>default value</i>	
<code>DataRow</code>	<code>List</code>	the function which is applied to each row of data in a file read by <code>ReadData</code>

## 6.2 Special graphics functions for Parameter Scans

A common way of displaying information from scans over the parameter space is either as a three-dimensional plot, or as a density plot. Usually the height of the figure (represented by levels of grey in the case of a density plot) then determines exclusion limits or similar information.

The standard *Mathematica* plotting functions `Plot3D` and `DensityPlot` require functions to be plotted, not arrays of numbers. Although this can be circumvented by using interpolating functions, there are two considerable disadvantages of this method: first, the grid used by the plotting function in general does not represent the calculated points<sup>§</sup>; second, there is no control over missing values.

For this reason, two graphics functions for 3D and density plots have been included in *FormCalc* in the `ScanGraphics` package. They are especially designed for parameter-scan plots in that they use a grid which precisely matches the computed values, and that they can deal with missing values.

```
ScanPlot3D[ $v_1$ ,  $v_2$ ,  $n$ , opts]
```

```
ScanDensityPlot[ $v_1$ ,  $v_2$ ,  $n$ , opts]
```

```
ScanContourPlot[ $v_1$ ,  $v_2$ ,  $n$ , opts]
```

make a 3D, density, or contour plot of a quantity which has been computed for different values of  $v_1$  and  $v_2$ , with  $v_1$  and  $v_2$  displayed on the  $x$ - and  $y$ -axes. The data to be plotted are assumed to be in the data sets `Data[1] . . . Data[ $n$ ]`. The argument *opts* may contain additional 3D (2D) graphics options.

Both functions cooperate closely with the `ReadData` function described in the last section. The argument  $n$  which specifies the number of data sets to be taken is usually just the output of `ReadData`.

<sup>§</sup>In fact, if the plotting function uses a grid which differs very slightly from the grid of data points, the plot may even display some funny bumps which are artifacts of the interpolation.

ScanPlot3D and ScanDensityPlot determine the grid on a democratic-vote basis: they make a list of all the spacings  $\{\Delta v_1, \Delta v_2\}$  between any two points and take the values of  $\Delta v_1$  and  $\Delta v_2$  that occur most often. (This of course assumes that the grid is equidistantly spaced.) Points on this grid which do not correspond to a data point are missing points. If missing points are detected, both graphics functions issue a warning and deposit the coordinates of the missing points in the variable `$MissingPoints` for checking.

## 7 Low-level functions for code output

*FormCalc*'s code-generation functions, used internally e.g. by *WriteSquaredME*, can also be used directly to write out an arbitrary Mathematica expression as optimized code. The basic syntax is very simple:

1. `handle = OpenCode["file.F"]`  
opens *file.F* as a Fortran file for writing,
2. `WriteExpr[handle, {var -> expr, ...}]`  
writes out Fortran code to calculate *expr* and store the result in *var*,
3. `Close[handle]`  
closes the file again.

The code generation is fairly sophisticated and goes well beyond merely applying Mathematica's *FortranForm*. The generated code is optimized, e.g. common subexpressions are pulled out and computed in temporary variables. Expressions too large for Fortran are split into parts, as in

```
var = part1
var = var + part2
...
```

If the expression is too large even to be sensibly evaluated in one file, the *FileSplit* function can distribute it on several files and optionally write out a master subroutine which calls the individual parts.

To further automate the code generation, such that the resulting code needs few or no changes by hand, many ancillary functions are available.

## 7.1 File handling, Type conversion

<code>MkDir[dirs]</code>	make sure the directory <i>dirs</i> exists, creating subdirectories as necessary
<code>OpenCode[file]</code>	open <i>file</i> for writing code
<code>TimeStamp[]</code>	return a string with the current date and time
<code>ToCode[expr]</code>	return a string with the Fortran (or C) form of <i>expr</i>
<code>ToSymbol[args]</code>	concatenate <i>args</i> into one symbol
<code>ToList[expr]</code>	return a list of summands of <i>expr</i>

`MkDir["dir1", "dir2", ...]` makes sure the directory *dir<sub>1</sub>/dir<sub>2</sub>/...* exists, creating the individual subdirectories *dir<sub>1</sub>, dir<sub>2</sub>, ...* as necessary. It works roughly as `mkdir -p` in Unix.

`OpenCode[file]` opens *file* for writing code.

`TimeStamp[]` returns a string with the current date and time.

`ToCode[expr]` returns the Fortran (or C) form of *expr* as a string.

`ToSymbol[args]` concatenates its arguments into a new symbol, e.g. `ToSymbol[a, 1, {x, y}]` gives `a1xy`.

`ToList[expr]` returns a list of summands of *expr*, i.e. turns a sum of items into a list of items.

## 7.2 Writing Expressions

<code>PrepareExpr[exprs]</code>	prepare <i>exprs</i> for write-out to code
<code>WriteExpr[file, exprs]</code>	write <i>exprs</i> to <i>file</i>
<code>CodeExpr[v, t, exprs]</code>	the expressions <i>exprs</i> with variables <i>v</i> and temporary variables <i>t</i> in a form ready for write-out by <code>WriteExpr</code>
<code>RuleAdd[a, b]</code>	same as <i>a</i> -> <i>a</i> + <i>b</i>
<code>DebugLine[s]</code>	a debugging statement for variable <i>s</i>
<code>\$SymbolPrefix</code>	a string prepended to all externally visible symbols to avoid symbol conflicts

`PrepareExpr[{var1 -> expr1, var2 -> expr2, ...}]` prepares a list of variable assignments for write-out to a code file. Expressions with a leaf count larger than `$BlockSize` are split into several pieces, as in

```
var = part1
```



```
var = var + part2
...
```

thereby possibly introducing temporary variables for subexpressions. `SumOver`, `DoLoop`, and `IndexIf` objects are properly taken into account as do-loops and if-statements. The output is a `CodeExpr[vars, tmpvars, exprlist]` object, where *vars* are the original and *tmpvars* the temporary variables introduced by `PrepareExpr`.

`WriteExpr[file, exprlist]` writes a list of variable assignments to *file*. The *exprlist* can either be a `CodeExpr` object, i.e. the output of `PrepareExpr`, or a list of expressions of the form  $\{var_1 \rightarrow expr_1, var_2 \rightarrow expr_2, \dots\}$ , which is first converted to a `CodeExpr` object using `PrepareExpr`. `WriteExpr` returns a list of the subexpressions that were actually written.

`CodeExpr[vars, tmpvars, exprlist]` is the output of `PrepareExpr` and contains a list of expressions ready to be written to a file, where *vars* are the original variables and *tmpvars* are temporary variables introduced in order to shrink individual expressions to a size small enough for Fortran.

`RuleAdd[var, expr]` is equivalent to  $var \rightarrow var + expr$ .

`DebugLine[s]` translates to a debugging statement for variable *s* (print-out of variable *s*) in the code.

`$SymbolPrefix` is a string prepended to all externally visible symbols in the generated code to avoid symbol collisions.

<i>PrepareExpr option</i>	<i>default value</i>	
Optimize	False	whether to introduce temporary variables for common subexpressions
Expensive	{}	objects which should be hoisted from inner loops as far as possible
MinLeafCount	10	the minimum leaf count above which a subexpression becomes eligible for abbreviation
DebugLines	False	whether to generate debugging statements for every variable
FinalTouch	Identity	a function which is applied to the final subexpressions
ResetNumbering	True	whether to number $\text{dup}_n$ and $\text{tmp}_n$ variables from $n = 1$

`Optimize` determines whether variables should be introduced for subexpressions which are used more than once.

`Expensive` lists patterns of objects which are expensive in terms of CPU time and should therefore be hoisted from inner do-loops if possible.

`MinLeafCount` specifies the minimum leaf count a common subexpression must have in order that a variable is introduced for it.

`DebugLines` specifies whether debugging statements are generated for each expression. If instead of `True` or `False` a string is given, debugging messages are generated and prefixed by this string. If a function is given, it is queried for each variable assignment and its output, `True`, `False`, or a string, individually determines generation of the debug statement. Debugging messages are usually generated for the expressions specified by the user only. To extend them to intermediate variables, e.g. the ones introduced for optimization, use `DebugLines`  $\rightarrow$  `All` (or `All[string]` or `All[func]`).

The actual debugging statements are constructed from the items in `$DebugCmd`, which is a list of four strings: The first two strings provide commands issued before and after the debugging statement, the second two strings provide the opening and closing of the debugging statement. The debugging statement has two arguments: a string (the variable name) and a number (the variable value). The default setting is

```
$DebugCmd = {"#ifdef DEBUG\n", "#endif\n", "DEB(", ")"};
```

Hence debugging is activated at compile time by defining the preprocessor variable `DEBUG` and the preprocessor macro `DEB` might be defined in Fortran as

```
#define DEB(a,x) print *, a, x
```

`FinalTouch` gives a function which is applied to each final subexpression, just before write-out to file.

`ResetNumbering` resets the internal numbering of variable names created by `PrepareExpr`, i.e. variables of the form `dup $n$`  (from `Optimize`) and `tmp $n$`  (from `Expensive`) start at  $n = 1$ . It may be necessary to disable this reset e.g. if more than one generated expression shall be put in the same program unit.

*Note:* these options may also be given to `WriteExpr` which simply hands them down to `PrepareExpr`. They cannot be set using `SetOptions[WriteExpr, ...]`, however.

<i>WriteExpr option</i>	<i>default value</i>	
<code>HornerStyle</code>	<code>True</code>	whether to order expressions according in Horner form
<code>FinalCollect</code>	<code>False</code>	whether to collect factors in the final expression
<code>FinalFunction</code>	<code>Identity</code>	a function to apply to the final expression before write-out
<code>Type</code>	<code>False</code>	the type of the expressions, <code>False</code> to omit declarations
<code>TmpType</code>	<code>"ComplexType"</code>	the type of temporary variables
<code>IndexType</code>	<code>False</code>	the type of indices
<code>RealArgs</code>	<code>{A0,A00,B0,B1, B00,B11,B001, B111,DB0,DB1, DB00,DB11, B0i,C0i,D0i, E0i,F0i,Bget, Cget,Dget,Eget, Log,Sqrt}</code>	functions whose arguments must be of a guaranteed type (default: real)
<code>Newline</code>	<code>""</code>	a string to be printed after each expression

`HornerStyle` specifies whether expressions are ordered in Horner form before writing them out as code.

`FinalCollect` chooses a final collecting of common factors, after going over to Horner form,

just before write-out to code.

`FinalFunction` specifies a function to be applied to the final expressions, just before write-out to code. This function can be used to apply language-specific translations.

`Type` determines whether declarations shall be generated for the variables and of which type. If a string is given, e.g. `Type -> "double precision"`, `WriteExpr` writes out declarations of that type for the given expressions. Otherwise no declarations are produced.

`TmpType` is the counterpart of `Type` for the temporary variables. `TmpType -> Type` uses the settings of the `Type` option.

`IndexType` likewise determines the generation of declarations for do-loop indices.

`RealArgs` gives a list of functions whose numerical arguments must be of a guaranteed type, usually real (double precision). For example, if the function *foo* expects a single real argument, it must be invoked as *foo*(0D0) in Fortran, not *foo*(0).

`RealArgs[foo] := ...` defines the actual conversion for *foo*. The default is to map `NArgs` over all arguments. `NArgs[args]` returns *args* with all integers turned into reals. Note that `NArgs` is not quite the same as `N`: while it changes 1 to 1., it leaves e.g. `m[1]` intact so that array indices remain integers.

`NewLine` specifies a string to be printed after each statement.

### 7.3 Variable lists and Abbreviations

<code>OnePassOrder[list]</code>	order <i>list</i> such that the definition of each item comes before its first use
<code>MoveDepsRight[r<sub>1</sub>, ..., r<sub>n</sub>]</code>	move variables among the lists <i>r<sub>1</sub>, ..., r<sub>n</sub></i> such that a definition does not depend on <i>r<sub>i</sub></i> further to the right
<code>MoveDepsLeft[r<sub>1</sub>, ..., r<sub>n</sub>]</code>	move variables among the lists <i>r<sub>1</sub>, ..., r<sub>n</sub></i> such that a definition does not depend on <i>r<sub>i</sub></i> further to the left

`OnePassOrder[r]` orders a list of interdependent rules such that the definition of each item (*var -> value*) comes before its use in the right-hand sides of other rules. When `OnePassOrder` detects a recursion among the definitions of a list, it deposits the offending rules in an internal format in `$OnePassDebug` as debugging hints.

`MoveDepsRight[r1, ..., rn]` shuffles variable definitions (*var -> value*) among the lists of rules *r<sub>i</sub>* such that the definitions in each list do not depend on definitions in *r<sub>i</sub>* further to the left. For example, `MoveDepsRight[{a -> b}, {b -> 5}]` produces `{ {}, {b -> 5, a -> b} }`, i.e. it moves *a -> b* to the right list because it depends on *b*.

`MoveDepsLeft[r1, ..., rn]` shuffles variable definitions (*var -> value*) among the lists of rules

$r_i$  such that the definitions in each list do not depend on definitions in  $r_i$  further to the right. For example, `MoveDepsLeft [{a -> b}, {b -> 5}]` produces `{ {b -> 5, a -> b}, {} }`, i.e. it moves `b -> 5` to the left list because that depends on `b`.

<code>SplitSums [expr]</code>	split <i>expr</i> into a list of expressions such that index summations apply to the whole of each part
<code>ToDoLoops [list]</code>	categorize <i>list</i> into patches that must be summed over the same set of indices
<code>DoLoop [expr, ind]</code>	a do-loop of <i>expr</i> over the indices <i>ind</i>

`SplitSums [expr]` splits *expr* into a list of expressions such that index sums (marked by `SumOver`) always apply to the whole of each part. `SplitSums [expr, wrap]` applies *wrap* to the coefficients of the `SumOver`.

`ToDoLoops [list, ifunc]` splits *list* into patches which must be summed over the same set of indices. *ifunc* is an optional argument: *ifunc* [*expr*] must return the indices occurring in *expr*.

`DoLoop [ind, expr]` is a symbol introduced by `ToDoLoops` indicating that *expr* is to be summed over the set of indices *ind*.

<code>ToIndexIf [expr]</code>	turn the <code>IndexDelta</code> and <code>IndexDiff</code> in <i>expr</i> into <code>IndexIf</code>
<code>IndexIf [cond, a, b]</code>	same as <code>If [cond, a, b]</code> except that the expressions <i>a</i> and <i>b</i> are not held unevaluated
<code>IndexDelta [i, j]</code>	Kronecker's $\delta_{ij}$
<code>IndexDiff [i, j]</code>	$1 - \delta_{ij}$
<code>MapIf [f, expr]</code>	maps <i>f</i> over the <i>expr</i> except for the conditional parts of an <code>IndexIf</code>

`ToIndexIf [expr]` converts all `IndexDelta` and `IndexDiff` objects in *expr* to `IndexIf`, which will be written out as if-statements in the generated code. `ToIndexIf [expr, patt]` operates only on indices matching *patt*.

`IndexIf [cond, a, b]` is the same as `If [cond, a, b]` except that expressions *a* and *b* are not held unevaluated. `IndexIf [cond, a]` is equivalent to `IndexIf [cond, a, 0]`, i.e. the “else” part defaults to 0. Several conditions can be combined as `IndexIf [cond1, a1, cond2, a2, ...]`, which is equivalent to `IndexIf [cond1, a1, IndexIf [cond2, a2, ...]]`. Despite its name, the statement is not restricted to index nor to integer comparisons. It is furthermore written out as a regular if statement, i.e.

```
if( cond ) then
  a
```

```

else
  b
endif

```

`IndexDelta[i, j]` is Kronecker's delta  $\delta_{ij}$ .

`IndexDiff[i, j]` is  $1 - \delta_{ij}$ .

`MapIf[f, expr]` is equivalent to `Map` except that it does not modify the conditional parts if `expr` is an `IndexIf`.

<code>BlockSplit[var -&gt; expr]</code>	split <code>expr</code> into subexpressions with leaf count less than <code>\$BlockSize</code>
<code>FileSplit[exprlist, mod, writemod, writeall]</code>	split <code>exprlist</code> into batches with leaf count less than <code>\$FileSize</code> , call <code>writemod</code> to write a each module to file and finally <code>writeall</code> to generate a 'master subroutine' which invokes the modules
<code>ToArray[s]</code>	take the symbol <code>s</code> apart into letters and digits, e.g. <code>X123</code> $\rightarrow$ <code>X[123]</code>
<code>Renumber[expr, v<sub>1</sub>, v<sub>2</sub>, ...]</code>	renumber all <code>v<sub>1</sub>[n]</code> , <code>v<sub>2</sub>[n]</code> , ... in <code>expr</code>
<code>MaxDims[args]</code>	find the maximum indices of all functions in <code>args</code>
<code>\$BlockSize</code>	the size of each block for <code>BlockSplit</code>
<code>\$FileSize</code>	the size of each file for <code>FileSplit</code>

`BlockSplit[var -> expr]` tries to split the calculation of `expr` into subexpressions each of which has a leaf count less than `$BlockSize`.

`FileSplit[exprlist, mod, writemod, writeall]` splits `exprlist` into batches with leaf count less than `$FileSize`. If there is only one batch, `writemod[batch, mod]` is invoked to write it to file. Otherwise, `writemod[batch, modN]` is invoked on each batch, where `modN` is `mod` suffixed by a running number, and in the end `writeall[mod, res]` is called, where `res` is the list of `writemod` return values. The optional `writeall` function can be used e.g. to write out a master subroutine which invokes the individual modules. If `mod` is given as a dot product `name.delim`, the delimiter is used to separate the `N` suffix. For example, `"foo"."_"` will evaluate to `"foo"` for a single file and to `"foo_1"`, `"foo_2"` etc. for multiple files.

`ToArray[s]` turns the symbol `s` into an array reference by taking it apart into letters and digits, e.g. `Var1234` becomes `Var[1234]`. `ToArray[expr, s1, s2, ...]` turns all occurrences of the symbols `s1NNN`, `s2NNN`, etc. in `expr` into `s1[NNN]`, `s2[NNN]`, etc.

`Renumber[expr, var1, var2, ...]` renumbers all `var1[n]`, `var2[n]`, ... in `expr`.

MaxDims[*args*] returns a list of all distinct functions in *args* with the highest indices that appear, e.g. MaxDims[foo[1, 2], foo[2, 1]] returns {foo[2, 2]}.

## 7.4 Declarations

SubroutineDecl[ <i>name</i> ]	the declaration for the subroutine <i>name</i>
VarDecl[ <i>vars</i> , <i>type</i> ] <div style="text-align: center;"><math>\underbrace{\hspace{1.5cm}}_a</math></div>	the declaration of <i>vars</i> as variables of type <i>type</i>
VarDecl[Common[ <i>com</i> ][ <i>a</i> ]]	the declaration of common block <i>com</i> with individual variables <i>a</i> ( <i>a</i> as above)
VarDecl[NameMap[ <i>com</i> ][ <i>a</i> ]]	the declaration of common block <i>com</i> with an array plus a preprocessor map of the <i>a</i> onto array elements ( <i>a</i> as above)
VarDecl[NotEmpty[ <i>a</i> ]]	output only if at least one variable list in <i>a</i> is non-empty
DoDecl[ <i>i</i> , <i>range</i> ]	the declaration of a do-loop of <i>i</i> over <i>range</i>
CallDecl[ <i>subs</i> ]	the invocations of subroutines <i>subs</i>
Dim[ <i>i</i> ]	the highest value the index <i>i</i> takes on

SubroutineDecl[*name*] returns a string with the declaration of the subroutine *name*.

VarDecl[{*v*<sub>1</sub>, *v*<sub>2</sub>, ...}, *t*] returns a string with the declaration of *v*<sub>1</sub>, *v*<sub>2</sub>, ... as variables of type *t*. Any other strings are output verbatim, e.g. "#ifdef COND\n", "#endif\n".

VarDecl[Common[*c*][{*v*<sub>1</sub>, *v*<sub>2</sub>, ...}, *t*]] declares *v*<sub>1</sub>, *v*<sub>2</sub>, ... to be members of common block *c*.

VarDecl[NameMap[*c*][{*v*<sub>1</sub>, *v*<sub>2</sub>, ...}, *t*]] works much like Common but puts only arrays into the common block (one for each data type), together with preprocessor statements mapping the *v*<sub>1</sub>, *v*<sub>2</sub>, ... onto array elements.

VarDecl arguments wrapped in NotEmpty are output only if at least one of its variable lists is non-empty.

DoDecl[*i*, *m*] returns a string with the declaration of a loop over *i* from 1 to *m*.

DoDecl[*i*, *a*, *b*] returns the same for a loop from *a* to *b*. DoDecl[*i*] invokes Dim[*i*] to determine the upper bound on *i*.

CallDecl[{*sub*<sub>1</sub>, *sub*<sub>2</sub>, ...}] returns a string with the invocations of the subroutines *sub*<sub>1</sub>, *sub*<sub>2</sub>, ..., taking into account possible loops indicated by DoLoop.

Dim[*i*] returns the highest value the index *i* takes on, as determined from the amplitude currently being processed.

## 7.5 Compatibility Functions

<code>ToOldBRules</code>	rules to convert to the conventions for two-point functions of <i>LoopTools</i> 2.2 and before
<code>ToNewBRules</code>	rules to convert to the conventions for two-point functions of <i>LoopTools</i> 2.3 or later

`ToOldBRules` and `ToNewBRules` are two sets of transformation rules that convert between the old  $(B_0, B_1, \dots)$  and new  $(B_{0i}[b_0, b_1, \dots])$  conventions for two-point functions in *LoopTools*.



## References

- [Ch79] M. Chanowitz, M. Furman, and I. Hinchliffe, *Nucl. Phys.* **B159** (1979) 225.
- [Cv76] P. Cvitanovic, *Phys. Rev.* **D14** (1976) 1536.
- [dA98] F. del Aguila, A. Culatti, R. Muñoz Tapia, and M. Pérez-Victoria, *Nucl. Phys.* **B537** (1999) 561 [hep-ph/9806451].
- [De93] A. Denner, *Fortschr. Phys.* **41** (1993) 307.
- [Ha98] T. Hahn and M. Pérez-Victoria, *Comp. Phys. Commun.* **118** (1999) 153 [hep-ph/9807565].
- [Ha00] T. Hahn, *Comp. Phys. Commun.* **140** (2001) 418 [hep-ph/0012260].
- [Ha02] T. Hahn, *Nucl. Phys. Proc. Suppl.* **116** (2003) 363 [hep-ph/0210220].
- [Ha04] T. Hahn, *Comp. Phys. Commun.* **168** (2005) 78 [hep-ph/0404043].
- [Ha04a] T. Hahn, *Nucl. Phys. Proc. Suppl.* **135** (2004) 333 [hep-ph/0406288].
- [Ha06] T. Hahn, physics/0607103.
- [Ni05] C.C. Nishi, *Am. J. Phys.* **73** (2005) 1160 [hep-ph/0412245].
- [Si79] W. Siegel, *Phys. Lett.* **B84** (1979) 193.
- [tH72] G. 't Hooft and M. Veltman, *Nucl. Phys.* **B44** (1972) 189.
- [Ve00] J.A.M. Vermaseren, math-ph/0010025. See also <http://www.nikhef.nl/~form>.
- [Ve96] J.A.M. Vermaseren, The use of computer algebra in QCD, in: H. Latal, W. Schweiger, Proceedings Schlading 1996, Springer Verlag, ISBN 3-540-62478-3.
- [WhBG05] M.R. Whalley, D. Bourilkov, R.C. Group, hep-ph/0508110.  
<http://hepforge.cedar.ac.uk/lhapdf/>.
- [Za03] A.F. Zarnecki, *Acta Phys. Polon.* **B34** (2003) 2741 [hep-ex/0207021].

## Acknowledgements

*FormCalc* would not be able to calculate in four dimensions without the hard work of Manuel Pérez-Victoria and his deep understanding of constrained differential renormalization.

The fermionic matrix elements would not be half as well implemented without the relentless testing of Christian Schappacher.

## Index

't Hooft–Veltman scheme, 13  
-2, 65  
-3, 65  
-o, 65  
.submitrc, 63  
\$AbbPrefix, 25  
\$BlockSize, 86  
\$DebugCmd, 82  
\$DriversDir, 42  
\$Editor, 13  
\$EditorModal, 13  
\$FileSize, 86  
\$FormAbbrDepth, 14  
\$FormCmd, 7  
\$MissingPoints, 79  
\$OptPrefix, 24  
\$SymbolPrefix, 80  
1to2.F, .h, 41, 53  
1to3.F, .h, 41  
2to2.F, .h, 41, 53  
2to3.F, .h, 41, 53  
2to4.F, .h, 41  
  
Abbr, 24  
abbr0\_cat, 49  
abbr1\_cat, 49  
abbri\_angle, 49  
abbri\_hel, 49  
abbri\_s, 49  
AbbrevDo, 25  
Abbreviate, 25  
abbreviations, 6, 23, 30, 84  
    registering, 27  
AbbrevSet, 25  
Alfa, 21  
Alfa2, 21  
Alfas, 21

Alfas2, 21  
All, 30, 32  
analytic amplitudes, 36  
antisymmetrization, 16  
Antisymmetrize, 15  
ApplyUnitarity, 38  
Automatic, 10  
  
bfunc.m, 41  
BlockSplit, 86  
Breitenlohner–Maison scheme, 13  
BREMSSTRAHLUNG, 54, 75  
bremsstrahlung, 74  
btensor.m, 41  
  
C code, 41  
C output, 44, 79, 80  
CA2, 22  
CalcFeynAmp, 8, 14  
CalcLevel, 10  
CalcRenConst, 73  
CallDecl, 87  
cancellation of terms, 11  
CB2, 22  
CBA2, 22  
CHARGE $i$ , 53  
Chiral, 11  
chirality projector, 20  
Classes, 10  
ClearProcess, 16  
code generation, 41  
code output, 79  
CodeExpr, 80  
colour indices, 19, 32  
COLOURFACTOR, 54  
ColourGrouping, 32  
ColourME, 32

ColourSimplify, 32  
 Combine, 16  
 Common, 87  
 compiling, 7  
     generated code, 58  
 components, 36  
 configure, 41, 58  
 constrained differential  
     renormalization, 6, 10  
 CUBA, 58  
 CUHRE, 58  
 CutMax, 55  
 CutMin, 55  
 CutTools, 12  
 Cvitanovic algorithm, 32  
 CW2, 21  
  
 Data, 77  
 data, 41, 64  
 data files, 63  
 DataHead, 67  
 DataRow, 69, 78  
 debugging options, 13  
 DebugLine, 80  
 DebugLines, 82  
 declarations, 87  
 DeclareProcess, 14  
 demo programs, 6  
 Den, 18  
 DenCollect, 38  
 denominator, 18  
 density matrix, 29  
 Deny, 26  
 dependences, 52  
 diagram generation, 7  
 DiagramType, 17  
 Dim, 87  
 Dimension, 10, 31, 35  
 dimension, 22

dimensional  
     reduction, 6, 10  
     regularization, 6, 10  
 Dirac equation, 8  
 Dirac matrix, 20  
 DiracChain, 20  
 Divergence, 37  
 DIVONNE, 58  
 Dminus4, 11, 31, 35, 37  
 Dminus4Eps, 31, 35  
 DoDecl, 87  
 DoLoop, 85  
 dot product, 18  
 DotExpand, 10, 31, 35  
 DottedSpinor, 21  
 driver program, 51  
 Drivers, 42  
 drivers, 41  
 DSelfEnergy, 72  
  
 $e[n]$ , 18  
 EditCode, 10, 31, 35  
 Eps, 18  
 ESOFTHMAX, 54, 75  
 Evanescent, 10  
 ExceptDirac, 15  
 ExpandSums, 38  
 expansion, 13  
 Expensive, 82  
 external fermions, 28  
 external spinors, 11  
 ExtraRules, 43  
  
 FERMION, 53  
 fermion chain, 11  
 fermion traces, 8  
 FermionChains, 10  
 fermionic matrix elements, 28  
 FermionicQ, 17  
 FermionOrder, 10

- FeynArts*, 7
- FieldRC, 72
- Fierz transformation, 11
- file handling, 80
- FileHeader, 43, 73
- FileIncludes, 43, 73
- FilePrefix, 43, 73
- FileSplit, 79, 86
- FileTag, 10
- FinalCollect, 83
- FinalFunction, 83
- FinalTouch, 82
- Finite, 18
- FIXED, 56
- Folder, 43, 46, 73
- FormAmp, 15
- FormDot, 14
- FormMat, 14
- FormPre, 14
- FormQC, 14
- FormQF, 14
- FormSub, 14
- Fortran code, 41
- Fortran output, 44, 79, 80
- four-vector, 36
- Fuse, 26
- Gamma5Test, 10
- Gamma5ToEps, 10
- GaugeTerms, 35
- generators of  $SU(N)$ , 19
- generic amplitude, 7
- gluon indices, 32
- gnuplot, 65
- Gram determinant, 18
- Hel[n], 30
- helicity, 29
- helicity matrix elements, 29
- helicity reference vector, 29
- HelicityME, 11, 14, 30
- HornerStyle, 83
- IDENTICALFACTOR, 54
- IGram, 18
- ImTilde, 72
- index contractions, 8
- IndexDelta, 85
- IndexDiff, 85
- IndexIf, 85
- IndexType, 83
- infrared divergences, 74
- InsertionPolicy, 10
- insertions, 12
- installation, 7
- integration, 54
- internal definitions, 16
- Invariants, 15
- invariants, 15
  - simplification, 16
- inverse Gram, 18
- invoking run, 59
- InvSimplify, 15
- k[n], 18
- Keep, 28
- kinematical simplification, 12
- kinematics, 16, 18
- language choice, 44
- leaf count, 24
- left-circular polarization, 59
- levels in diagrams, 7, 10
- Levi-Civita tensor, 18
- LIBS, 45
- local terms, 8
- log directory, 63
- log files, 63
- LogFile, 67
- longitudinal polarization, 59

- loop integral
  - symmetrized, 12
- LoopSquare, 43
- looptools.h, 53
- Lor[ $n$ ], 20
- Lorentz index, 20
- Lower, 55
- LScalarCoeff, 72
- lumi\_\*.F, 41
- lumi\_hadron.F, 53
- lumi\_parton.F, 53
- lumi\_photon.F, 53
- LVectorCoeff, 72
- MA02, 22
- main.F, .h, 41, 51, 53
- make, 58
- makefile, 45, 58
- makefile.in, 41
- Mandelstam variables, 15, 19, 23
- MapIf, 85
- mass shell, 15
- MASSi, 53
- MassDim, 22
- MassDim0, 22
- MassDim1, 22
- MassDim2, 22
- MassRC, 72
- Mat, 30, 32, 33
- MB2, 21
- MC2, 21
- MCha2, 22
- MD2, 21
- ME2, 21
- MG02, 22
- MG12, 22
- MGp2, 22
- Mh02, 22
- MH2, 21
- MHH2, 22
- MHp2, 22
- MinLeafCount, 26, 82
- missing points, 79
- MkDir, 80
- mktm, 41
- ML2, 21
- MLE2, 21
- MM2, 21
- MmaGetComplex, 66
- MmaGetComplexList, 66
- MmaGetInteger, 66
- MmaGetIntegerList, 66
- MmaGetReal, 66
- MmaGetRealList, 66
- MmaPutComplex, 69
- MmaPutComplexList, 69
- MmaPutInteger, 69
- MmaPutIntegerList, 69
- MmaPutReal, 69
- MmaPutRealList, 69
- MNeu2, 22
- model parameters, 21
- model-dependent symbols, 23
- model\_\*.F, 41
- model\_mssm.F, 47, 53
- model\_sm.F, 47, 53
- model\_thdm.F, 47, 53
- ModelConstIni, 47
- ModelDigest, 47
- ModelVarIni, 47
- MomElim, 10, 31, 35
- momenta, 18
- momentum conservation, 11
- MomRules, 10
- MoveDepsLeft, 84
- MoveDepsRight, 84
- MQD2, 21
- MQU2, 21

- MS2, 21
- MSf2, 22
- MSSMSimplify, 22
- MT2, 21
- MU2, 21
- MultiplyDiagrams, 17
- MW2, 21
- MZ2, 21
  
- NameMap, 87
- NArgs, 84
- Neglect, 22
- neglecting masses, 22
- Newline, 83
- NoBracket, 10, 35
- NoCostly, 10
- NoExpand, 10
- non-expansion of terms, 13
- normalization, 15
- Normalized, 15
- NotEmpty, 87
- NPID, 46
- numbering of momenta, 18, 53
- numerator function, 12
- numerical evaluation, 6, 39
- numerical parameters, 47
  
- OffShell, 17
- on-shell particles, 15
- OnePassOrder, 84
- OnShell, 15
- OnSize, 38
- open fermion chains, 8
- OpenCode, 80
- OPP, 12
- OPP, 10
- OPPQSlash, 10
- Optimize, 82
- OptimizeAbbr, 24
- ordering, 11
  
- Pair, 18
- Para, 77
- ParaHead, 67
- parallelization, 60
- parameter scan, 59
- parameter space, 61
- parameters, 47
- Particles, 10
- PARTON1, 2, 46
- partonic process, 45
- partonic.h, 46
- PaVeReduce, 10
- PDG code, 46
- permissions, 64
- phase space, 54
- PHOTON, 53
- photon mass, 75
- PID, 46
- pnuglot, 41, 65
- polarization, 59
- polarization sum, 34
- polarization vector, 15, 18
- PolarizationSum, 14, 34
- Pool, 38
- post-processing, 75
- PostFunction, 10
- PreFunction, 10
- PrepareExpr, 80
- Preprocess, 26
- process definition, 53
- process.h, 41, 51, 53
- ProcName, 46
- propagator denominator, 18
  
- rational terms, 12
- RCInt, 74
- RCSUB, 74
- ReadData, 76
- ReadData.tm, 41

- ReadFormDebug, 13
- RealArgs, 83
- RegisterAbbr, 27
- registering abbreviations, 27
- RegisterSubexpr, 27
- regularization method, 10
- RenConst, 71
- renconst.h, 53, 73
- renormalization constants, 71
- Renumber, 86
- reorder.c, 41
- ResetNumbering, 82
- resume, 63
- RetainFile, 10, 31, 35
- ReTilde, 72
- right-circular polarization, 59
- RScalarCoeff, 72
- RuleAdd, 80
- run.F, 41, 51, 53, 58
- RVectorCoeff, 72
- S, 19
- s[n], 18, 30
- SA2, 22
- Samurai, 12
- SB2, 22
- SBA2, 22
- SCALAR, 53
- ScanContourPlot, 78
- ScanDensityPlot, 78
- ScanGraphics.m, 41
- ScanPlot3D, 78
- scans, 61
- selecting diagrams, 17
- SelfEnergy, 72
- serial number, 59
- SetLanguage, 44
- SetNumber, 67
- SetupCodeDir, 41
- sfx, 41, 64
- short-hands, 23
- Show, 55
- Sij, 19
- SimplifyQ2, 10
- size of the amplitude, 24
- SMSimplify, 21
- soft-photon factor, 74
- SortDen, 10
- specs.h, 46
- spin, 29
- Spinor, 20, 21
- spinor chain, 11, 28
- Split, 54
- SplitSums, 85
- squared matrix element, 33
- SquaredME, 33, 49
- squaredme.a, 59
- SquaredME.F, 51
- Step, 55
- SU(N) objects, 19, 32
- SUAVE, 58
- SubExpr, 27
- subexpressions
  - registering, 27
- submit, 41, 63
- SubroutineDecl, 87
- SubroutineIncludes, 43, 73
- SumLegs, 35
- SUNN, 19
- SUNT, 19, 32
- SUSYTrigExpand, 22
- SUSYTrigReduce, 22
- SUSYTrigSimplify, 22
- SW2, 21
- SymbolPrefix, 43, 46, 73
- T, 19
- TadpoleRC, 72

- TagCollect, 38
- TagDiagrams, 17
- TB2, 22
- tensor coefficients, 12
- tensor reduction, 8
- test,  $\gamma_5$ , 12
- threshold, 59
- $T_{ij}$ , 19
- TimeStamp, 80
- TmpType, 83
- ToArray, 86
- ToCode, 80
- ToComponents, 36
- ToDoLoops, 85
- ToIndexIf, 85
- ToList, 80
- ToNewBRules, 88
- ToOldBRules, 88
- tools, 41
- ToSymbol, 80
- transversality, 15
- Transverse, 15
- TreeSquare, 43
- TRIVIAL, 56
- turnoff, 41, 64
- Type, 83
- type conversion, 80
- TYPEi, 53
- U, 19
- UChC, 22
- ultraviolet divergences, 37
- unpolarized case, 29
- unpolarized particles, 59
- unsquared, 36
- Upper, 55
- useful functions, 38
- USfC, 22
- util.a, 45
- UV divergences, 10
- UVDivergentPart, 37
- UVSeries, 37
- VA, 11
- Var, 55
- VarDecl, 87
- variable lists, 84
- VChC, 22
- VecSet, 36
- VECTOR, 53
- vector, 36
- vectorization, 60
- VEGAS, 58
- Weyl, 11
- WeylChain, 21
- WF\_RENORMALIZATION, 54
- WidthRC, 72
- WriteExpr, 80
- WriteRenConst, 41, 73
- WriteSquaredME, 41
- writing expressions, 80
- xsection.F, 51
- xsection.F, .h, 41
- ZNeuC, 22