FormCalc 9.6 User's Guide

Sep 29, 2019 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.

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` 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`

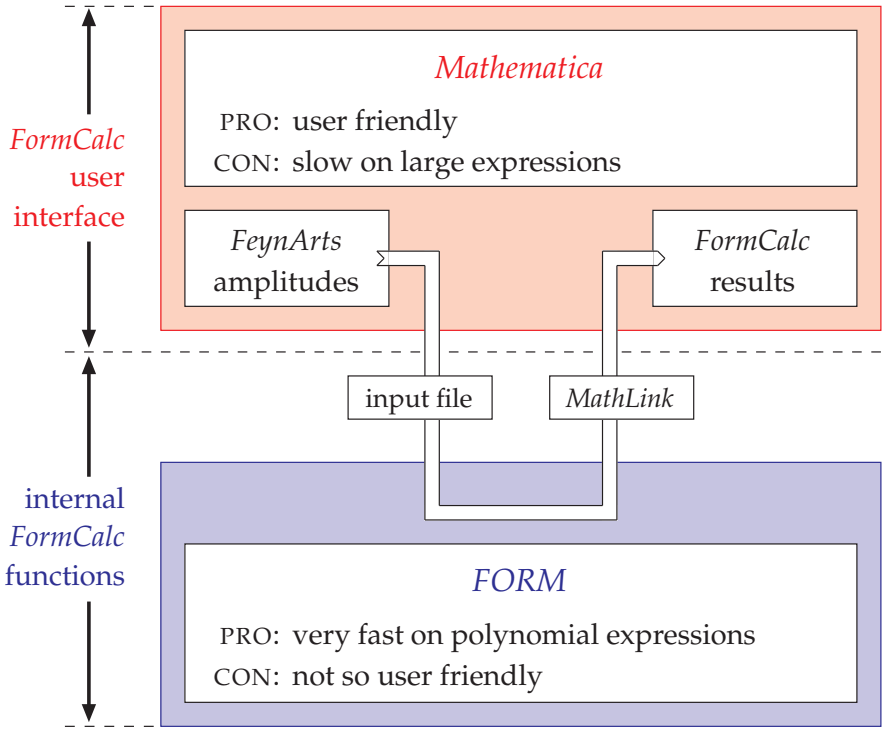
Contents

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*:

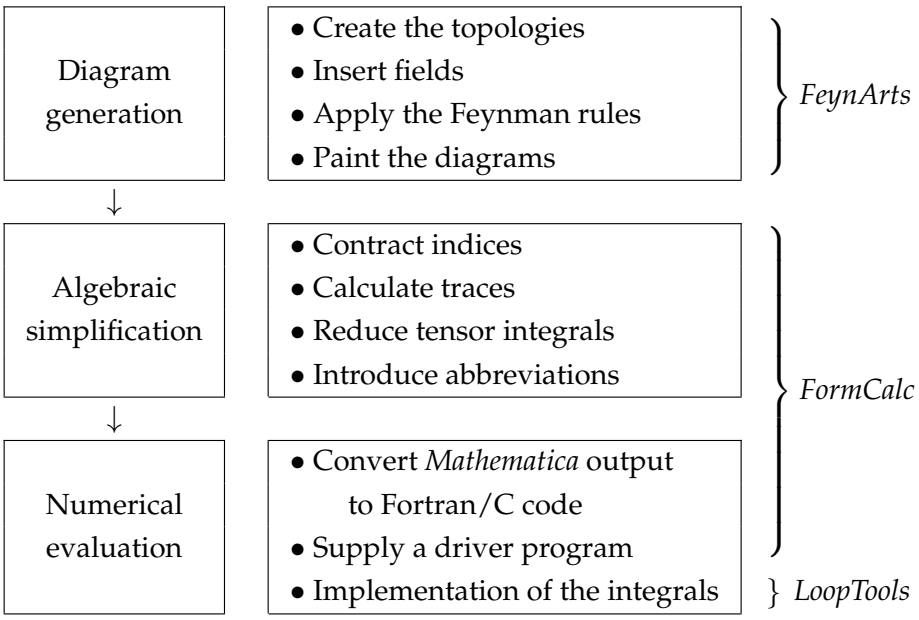


FormCalc combines the speed of *FORM* with the powerful instruction set of *Mathematica* and the latter greatly facilitates further processing of the results. Owing to *FORM*'s speed, *FormCalc* can process, for example, the 1000-odd one-loop diagrams of W–W scattering in the Standard Model in a few 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-x86-64`. A single *FormCalc* installation can thus be NFS-mounted on different systems once `compile` has been run on each.

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. 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[a_1, a_2, ...]	calculate the sum of amplitudes $a_1 + a_2 + \ldots$

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*][r_1, r_2, ...]

*In D dimensions, the divergent integrals are expanded in $\varepsilon = (4 - D)/2$ up to order ε^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, ..., 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:

option	*default value*	
`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`)
`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
`SortDen`	`True`	whether to sort the denominators of the loop integrals
`PaVeReduce`	`False`	whether to analytically reduce tensor to scalar integrals

option	*default value*	
`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
`OPPQSlash`	`False`	whether to introduce $\tilde{\mu}$ also on $\not{q}$
`Gamma5Test`	`False`	whether to substitute $\gamma_5 \to \gamma_5(1 + \texttt{Gamma5Test}(D-4))$
`Gamma5ToEps`	`False`	whether to substitute $\gamma_5 \to \frac{1}{4!}\varepsilon_{\mu\nu\rho\sigma}\gamma^\mu\gamma^\nu\gamma^\rho\gamma^\sigma$ 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 (γ_μ, $g_{\mu\nu}$, 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 γ_5 to D dimensions is problematic. *FormCalc* employs a naive scheme [Ch79] and works with an anticommuting γ_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`.

`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].

The `FermionOrder` option means different things for Dirac and for Weyl chains:

For Dirac spinor chains (`FermionChains -> Chiral` or `VA`) `FermionOrder` determines the ordering of the external fermions within the chains. Choices 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).

For Weyl spinor chains (`FermionChains -> Weyl`) `FermionOrder` determines the structuring of the amplitude with respect to the fermion chains: Setting `FermionOrder -> Mat` wraps the Weyl fermion chains in `Mat`, like their Dirac counterpart, so that they end up at the outer-

most level of the amplitude and their coefficients can be read off easily. The matrix elements of the form `Mat[Fi, Fj]` are computed with the `WeylME` function.

The `Evanescent` option toggles whether fermionic operators are tracked across Fierz transformations by emitting terms of the form `Evanescent[`*original operator*`,` *Fierzed 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: $N_2/(D_0D_1) + N_3/(D_0D_1D_2) \to (N_2D_2 + N_3)/(D_0D_1D_2)$.

OPP specifies an integer N starting from which an N-point function is treated with OPP methods. For example, `OPP -> 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 $\not{q} \to \not{q} + \mathrm{i}\gamma_5\tilde{\mu}$, where odd powers of $\tilde{\mu}$ are eventually set to zero.

`Gamma5Test -> True` substitutes each γ_5 by $\gamma_5(1 + \texttt{Gamma5Test}(D-4))$ and it can be tested whether the final result depends on the variable `Gamma5Test` (which it shouldn't).

`Gamma5ToEps -> True` substitutes all γ_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 γ_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. `k1`, ..., `kn` for the momenta, `e1`, ..., `en` for the polarization vectors.

`PreFunction` and `PostFunction` specify functions to be applied to the amplitude before and after all simplifcations 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`.

`FormPre[`*amp*`]`	a function to set up the following `Form*` simplification functions, *amp* is the raw amplitude
`FormSub[`*subexpr*`]`	a function applied to subexpressions extracted by FORM
`FormDot[`*dotprods*`]`	a function applied to combinations of dot products by FORM
`FormMat[`*matcoeff*`]`	a function applied to the coefficients of matrix elements (`Mat[...]`) in the FORM output
`FormNum[`*numfunc*`]`	a function applied to numerator functions in the FORM output (OPP only)
`FormQC[`*qcoeff*`]`	a function applied to loop-momentum-independent parts of the OPP numerator in the FORM output
`FormQF[`*qfunc*`]`	a function applied to loop-momentum-dependent parts of the OPP numerator in the FORM output
`$FormAbbrDepth`	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`. Alternately, one can use *FormCalc*'s `Profile` function to narrow down performance problems, as in: `FormMat = Profile[FormMat]`.

4.2 `DeclareProcess`

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

`DeclareProcess[`a_1`,` a_2`, ...]`	set up internal definitions for the calculation of the amplitudes $a_1, a_2, \ldots$

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.

option	*default value*	
OnShell	True	whether the external momenta are on shell, i.e. $k_i^2 = m_i^2$
Invariants	True	whether to introduce kinematical invariants like the Mandelstam variables
Transverse	True	whether polarization vectors should be orthogonal to the corresponding momentum, i.e. $\varepsilon_i^\mu k_{i,\mu} = 0$
Normalized	True	whether the polarization vectors should be assumed normalized, i.e. $\varepsilon_i^\mu \varepsilon_{i,\mu}^* = -1$
InvSimplify	True	whether to simplify combinations of invariants as much as possible
MomElim	Automatic	how to apply momentum conservation
DotExpand	False	whether to expand terms collected for momentum elimination
Antisymmetrize	True	whether Dirac chains shall be 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 ith 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 $2 \to 2$ 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 ε_i are the ith 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`	$k_i \cdot k_j = \pm\frac{1}{2}\left[(s\|t)_{ij} - m_i^2 - m_j^2\right]$
`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.

`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, 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.

`Antisymmetrize` determines whether to antisymmetrize Dirac chains. This does not affect Weyl chains, i.e. has an effect only together with `FermionChains -> Chiral` or `VA`. Antisymmetrized chains carry a negative chirality identifier, e.g. `DiracChain[-6, μ, ν]` stands for $P_R \frac{1}{2}(\gamma_\mu\gamma_\nu - \gamma_\nu\gamma_\mu)$.

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_1`,`amp_2`,...]`	combines $amp_1, amp_2, \ldots$

The following two functions are helpful to select diagrams.

`FermionicQ[`d`]`	`True` if diagram d contains fermions
`DiagramType[`d`]`	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 $2 \to 2$ 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`.

`OffShell[`amp`, `i` -> `μ_i`, ...]`	enforce the relation $p_i^2 = \mu_i^2$ on the amplitudes amp.

`OffShell[`amp`, `i` -> `μ_i`, `j` -> `μ_j`, ...]` takes legs $i, j, \ldots$ off-shell by substituting the true on-shell relation $p_i^2 = m_i^2$ by $p_i^2 = \mu_i^2$. This is different from setting 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.

`MultiplyDiagrams[`f`][`$amps$`]`	multiply the diagrams in $amps$ with the factor returned by the function f
`TagDiagrams[`amp`]`	multiply each diagram in amp with the identifier `Diagram[`n`]`, 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 is 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]`...`k[`n_{in}`]` and the outgoing momenta `k[`n_{in} `+ 1]`...`k[`n_{in} `+` n_{out}`]`.

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

Note the extra factor $-\mathrm{i}$ 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^μ_μ from a self-contracted coupling), local terms arise because the ε 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 finiteness of an amplitude is to substitute the loop integrals by their divergent parts and test whether the coefficients of ε^{-1} and ε^{-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.

`S`	Mandelstam variable $s = (\texttt{k[1]} + \texttt{k[2]})^2$
`T`	Mandelstam variable $t = (\texttt{k[1]} - \texttt{k[3]})^2$
`U`	Mandelstam variable $u = (\texttt{k[2]} - \texttt{k[3]})^2$
`S`ij	invariant of the invariant-mass type, `S`$ij = (\texttt{k[}i\texttt{]} + \texttt{k[}j\texttt{]})^2$
`T`ij	invariant of the momentum-transfer type, `T`$ij = (\texttt{k[}i\texttt{]} - \texttt{k[}j\texttt{]})^2$

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

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

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

variable	*default value*	
`SUNN`	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 γ_μ `k[1]`$^\mu$ inside a `DiracChain`.
inside a `DiracChain`:	
`1`	$\mathbb{1}$
`5`	γ_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 `DiracChain[-1,`μ`,`ν`]` $= \sigma_{\mu\nu}$. Note that the antisymmetrization does not extend to the chirality projector or γ_5, e.g.

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

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 σ_μ`k[1]`$^\mu$ inside a `WeylChain`.
inside a `WeylChain`:	
`6`	signifies that the following sigma matrix has upper spinor indices
`7`	signifies that the following sigma matrix has lower spinor indices
`Spinor[`p`,` m`,` s`,` d`,` e`]`	a 2-spinor, with $d = 1(2)$ undotted (dotted) and $e = 0(1)$ uncontracted (contracted) with $\varepsilon = \left(\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right)$

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. `MW2` is an ordinary symbol while `MW^2` is a non-atomic expression (the internal representation is `Power[MW, 2]`).

Setting `$NoModelSpecific = True` before loading FormCalc inhibits setting of the model-dependent symbols. To change or add to these definitions, edit `FormCalc/ModelSpecific.m`. Currently, the following model-dependent symbols are defined:

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[`*expr*`]`	`Simplify` with `SW2 -> 1 - CW2` and `CW2 -> MW2/MZ2`

for the MSSM:	
`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`, `UChaC`, `VChaC`, `ZNeuC`	the complex conjugates of various mixing matrix elements
`MGl2`, `MSf2`, `MCha2`, `MNeu2`	the squares of various masses
`Mh02`, `MHH2`, `MA02`	
`MG02`, `MHp2`, `MGp2`	
`MSSMSimplify[`*expr*`]`	`Simplify` with SUSY trigonometric identities (e.g. `SBA2 -> 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.

`Neglect[`*s*`] = 0`	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 $\texttt{Mass}^2$ 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 $(\textit{number}) \cdot \texttt{Mass}^n$.

`MassDim[`*expr*`]`	replace the `MassDim`*n*-symbols in *expr* by $(\textit{random number}) \cdot \texttt{Mass}^n$
`MassDim0`	a list of symbols of mass dimension 0
`MassDim1`	a list of symbols of mass dimension 1
`MassDim2`	a list of symbols of mass dimension 2
`Mass`	a symbol representing the mass dimension

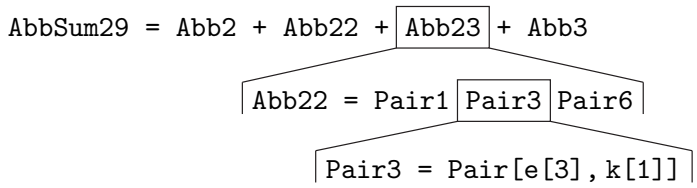
4.5 Handling Abbreviations

`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

```
C0i[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 `F`*n*, 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.

`Abbr[]`	the list of abbreviations introduced so far
`Abbr[`*patt*`]`	the list of all abbreviations including (or excluding, if preceded by `!` (`Not`)) the pattern *patt*
`Unabbr[`*expr*`]`	substitute back all abbreviations and subexpressions
`Unabbr[`*expr*`,` *patt*`]`	substitute back only those free of *patt*
`Unabbr[`*expr*`, !`*f* `[,` *patt*`]]`	do not substitute inside objects matching *f*
`OptimizeAbbr[`*a*`]`	optimize the abbreviations *a*
`$OptPrefix`	the prefix for additional abbreviations introduced by `OptimizeAbbr`
`SubstAbbr[`*exprlist*`,` *patt*`]`	expand out abbreviations matching *patt* in *exprlist*
`SubstSimpleAbbr[`*exprlist*`]`	expand out 'simple' abbreviations in *exprlist*

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 `Opt`*N* in the example above, can be chosen through the global variable `$OptPrefix`.

`SubstAbbr[`*exprlist*`,` *patt*`]` expands out all abbreviations matching *patt* in *exprlist*; e.g. if the abbreviation *a* `->` *b* matches, the definition *a* `->` *b* is removed and all *a* in *exprlist* are substituted by *b*, and the definition *a* `->` *b*.

`SubstSimpleAbbr[`*exprlist*`]` expands out 'simple' abbreviations in *exprlist*. Abbreviations are 'simple' if they are of the form $(number)$ or $(number) \cdot (symbol)$, or if the rhs's LeafCount is not larger than the lhs's. 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.

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

`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.

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

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.

Subexpr[]	the list of abbreviations introduced for subexpressions so far
Subexpr[*args*]	executes Abbreviate[*args*] 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.

`RegisterAbbr[`*abbr*`]`	register a list of abbreviations
`RegisterSubexpr[`*subexpr*`]`	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.

`Keep[`*expr*`,` *name*`,` *path*`]`	loads *path*`/`*name*`.m` if it exists, otherwise evaluates *expr* and stores the result (together with the output of `Abbr[]` and `Subexpr[]`) in that file. *path* is optional and defaults to `$KeepDir`
`Keep[`*expr*`,` *name*`]`	same as `Keep[`*expr*`,` *name*`,` `$KeepDir]`
`Keep[`*lhs* `=` *rhs*`]`	same as *lhs* `=` `Keep[`*rhs*`,` `"`*lhs*`"]`
`$KeepDir`	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 `F`i, 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 $2\,\mathrm{Re}(\mathcal{M}_0^*\mathcal{M}_1)$ only products $(\mathtt{F}i\,\mathtt{F}j^*)$ of spinor chains appear and these can be calculated using the density matrix for spinors

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

where $\lambda = \pm 1$ 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 λ-dependent part adds up to zero, so the projectors become

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

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 the helicity matrix elements $(\mathtt{F}i\,\mathtt{F}j^*)$.

`HelicityME[`$\mathcal{M}_0$`,` $\mathcal{M}_1$`]`	calculate the helicity matrix elements for all combinations of `F`n appearing in $\mathcal{M}_0^* M_1$
`All`	(used as either argument of `HelicityME`:) instead of selecting the `F`s which appear in an expression, simply take all `F`s currently in the abbreviations
`Mat[F`i`, F`j`]`	the helicity matrix element resulting from the spinor chains in `F`i and `F`j
`Hel[`n`]`	the helicity λ_n of the nth external particle
`s[`n`]`	the helicity reference vector of the nth external particle

To be sure, `HelicityME` does not calculate the full expression $\mathcal{M}_0^*\mathcal{M}_1$, only the combinations of `F`s that appear in this product. These are 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 $\mathcal{M}_0^*\mathcal{M}_1$.

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[...]
```

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

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

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.

option	*default value*	
`Dimension`	`4`	the dimension to compute in
`TreeSquare`	`$TreeSquare` (default: `True`)	whether to include the matrix elements for the computation of $\|\mathcal{M}_0\|^2$
`LoopSquare`	`$LoopSquare` (default: `False`)	whether to include the matrix elements for the computation of $\|\mathcal{M}_1\|^2$
`RetainFile`	`False`	whether to retain the temporary *FORM* command file
`EditCode`	`False`	whether to display the *FORM* code in an editor before sending it to *FORM*

`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 $D \to 4$ 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.

The `TreeSquare` and `LoopSquare` options govern whether, in addition to $\mathcal{M}_0^*\mathcal{M}_1$, also $|\mathcal{M}_0|^2$ and/or $|\mathcal{M}_1|^2$ will be needed. This allows to obtain all helicity matrix elements for e.g. the computation of $|\mathcal{M}_0|^2 + 2\,\mathrm{Re}\,\mathcal{M}_0^*\mathcal{M}_1$ with a single invocation of `HelicityME`. The selections are stored in the global variables `$TreeSquare` (`True` by default) and `$LoopSquare` (`False` by default) and used as defaults in further invocations of `HelicityME`, `ColourME`, `WeylME`, and `WriteSquaredME`.

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

The matrix element method can be applied to Weyl chains, too, which makes sense if one wishes to apply correction factors to the matrix elements. To this end `CalcFeynAmp` must be instructed to encode the Weyl fermion chains `F`i as matrix elements `Mat[F`i`]` (and sort the

amplitude accordingly) with the option `FermionOrder -> Mat`. The corresponding bilinear matrix elements `Mat[F`i`, F`j`]` needed to compute the squared amplitude are then computed with `WeylME`.

`WeylME[`$\mathcal{M}_0$`, `$\mathcal{M}_1$`]`	calculate the fermion matrix elements for all combinations of `F`n appearing inside `Mat` in $\mathcal{M}_0^*\mathcal{M}_1$
`All`	(used as either argument of `WeylME`:) instead of selecting the `F`s which appear in an expression, simply take all `F`s currently in the abbreviations

Since about the only purpose of Weyl matrix elements is to insert correction factors in front of the `Mat[F`i`, F`j`]`, there is an explicit option for this.

option	*default value*	
`MatFactor`	`(1 &)`	a function f which determines the correction factor f`[F`i`, F`j`]` multiplied with `Mat[F`i`, F`j`]`
`TreeSquare`	`$TreeSquare` (default: `True`)	whether to include the matrix elements for the computation of $\lvert\mathcal{M}_0\rvert^2$
`LoopSquare`	`$LoopSquare` (default: `False`)	whether to include the matrix elements for the computation of $\lvert\mathcal{M}_1\rvert^2$

4.9 Colour Matrix Elements

Diagrams involving quarks or gluons usually[‡] 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^a_{ij} 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 `SUNT[`a`, `b`, ... , `i`, `j`]` $= (T^a T^b \cdots)_{ij}$ where i and j are the external colour indices and the a, b, ... are the indices of external gluons. This notation includes also the identity in colour space as the special case with no external gluons: δ_{ij} = `SUNT[`i`, `j`]`.
- Traces over products of generators; these are denoted by `SUNT[`a`, `b`, ... , 0, 0]` $=$ $\mathrm{Tr}(T^a T^b \cdots)$.

[‡]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 SUNn. 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 SUNi and SUNj. (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 SU(N) case!

Because of this close analogy, the combinations of SUNi and SUNj are called colour matrix elements in *FormCalc* and are written accordingly as Mat[SUNi, SUNj]. The function which computes them is ColourME. It is invoked just like HelicityME.

ColourME[$\mathcal{M}_0$, $\mathcal{M}_1$]	calculate the colour matrix elements for all combinations of SUNn appearing in $\mathcal{M}_0^*\mathcal{M}_1$
All	(used as either argument of ColourME:) instead of selecting the SUNs which appear in an expression, simply take all SUNs currently in abbreviations
Mat[SUNi, SUNj]	the colour matrix element resulting from the SU(N) objects SUNi and SUNj

option	*default value*	
TreeSquare	$TreeSquare (default: True)	whether to include the matrix elements for the computation of $\lvert\mathcal{M}_0\rvert^2$
LoopSquare	$LoopSquare (default: False)	whether to include the matrix elements for the computation of $\lvert\mathcal{M}_1\rvert^2$

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

ColourSimplify[*expr*]	simplify colour objects in *expr*
ColourSimplify[*tree*, *loop*]	simplifies the colour objects in (*tree** *loop*)

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*.

ColourGrouping[*ins*]	group the inserted topologies (output of InsertFields) 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 $\mathcal{M}_0^*\mathcal{M}_1$.

This is non-trivial only if there are matrix elements of the form Mat[i, j] around, which have to be put in the right places. Specifically, if $\mathcal{M}_0$ and $\mathcal{M}_1$ are written in the form

$$\mathcal{M}_0 = a_{11}\,\mathtt{F1\,SUN1} + a_{21}\,\mathtt{F2\,SUN1} + \ldots = \sum_{ij} a_{ij}\,\mathtt{F}i\,\mathtt{SUN}j \quad \text{and}$$
$$\mathcal{M}_1 = b_{11}\,\mathtt{F1\,SUN1} + b_{21}\,\mathtt{F2\,SUN1} + \ldots = \sum_{ij} b_{ij}\,\mathtt{F}i\,\mathtt{SUN}j\,,$$

their product becomes

$$\begin{aligned}\mathcal{M}_0^*\mathcal{M}_1 \;&=\; a_{11}^* b_{11}\,\mathtt{Mat[F1,F1]\,Mat[SUN1,SUN1]} + \\ &\quad\; a_{21}^* b_{11}\,\mathtt{Mat[F1,F2]\,Mat[SUN1,SUN1]} + \ldots \\ &=\; \sum_{ijk\ell} a_{ik}^* b_{j\ell}\,\mathtt{Mat[F}j\mathtt{,\,F}i\mathtt{]\,Mat[SUN}\ell\mathtt{,\,SUN}k\mathtt{]}\,.\end{aligned}$$

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 M0)
      do 1 j = 1, (# of Fs in M1)
      do 1 k = 1, (# of SUNs in M0)
      do 1 l = 1, (# of SUNs in M1)
        result = result + Conjugate(a(i,k))*b(j,l)*MatF(j,i)*MatSUN(l,k)
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[$\mathcal{M}_0$, $\mathcal{M}_1$]	calculates $\mathcal{M}_0^*\mathcal{M}_1$, taking care to put the 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.

For clarity of output the result of SquaredME is given as two pieces: {*expr*, *rul*}, where *expr* is the squared amplitude expressed in terms of form factors FF and FFC, and *rul* is a list of rules which provide the values of the FF and FFC.

`SquaredME` does not insert the actual values for the `Mat[`i`,`j`]`. This can easily be done later by applying the output of `HelicityME`, `ColourME`, or `WeylME`, which are lists of rules substituting the `Mat[`i`,`j`]` by their values. That is to say, `SquaredME` and `HelicityME/ColourME/WeylME` perform complementary tasks: the former builds up the squared amplitude in terms of the `Mat[`i`,`j`]` whereas the latter calculate the `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

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

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

In the massless case the polarization sum is gauge dependent and η is an external four-vector which fulfills $\eta\cdot\varepsilon = 0$ and $\eta\cdot k \neq 0$. *FormCalc* makes the additional assumption $\eta\cdot\eta = 0$ and also drops the η^2 term above. For a gauge-invariant quantity, the η-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. Alternately, *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).

option	*default value*	
SumLegs	All	which external legs to include in the polarization sum
Dimension	4	the dimension to compute in
GaugeTerms	True	whether to retain η-dependent terms
NoBracket	(taken from CalcFeynAmp)	symbols not to be included in the bracketing in FORM
RetainFile	False	whether to retain the temporary *FORM* command file
EditCode	False	whether to display the *FORM* code in an editor before sending it to *FORM*

SumLegs allows to restrict the polarization sum to fewer than all external vector bosons. For example, SumLegs -> {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 $D \to 4$ 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 η. More precisely, the η-terms are actually introduced at first, to let potential cancellations of η'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}]
```

`VecSet[`n`,` m`,` p`,` $\{e_x, e_y, e_z\}$`]`	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`.

`ToComponents[`amp`,` $p_1 p_2 \ldots p_n$`]`	
`ToComponents[`amp`,` $\{p_1, p_2, \ldots, p_n\}$`]`	evaluate amp by substituting four-vectors by their component-wise representation using external polarizations $p_1, \ldots, p_n$

`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.

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

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

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

4.14 Useful Functions

`DenCollect[`*expr*`]`	collects terms in *expr* whose denominators are identical up to a numerical constant
`DenCollect[`*expr*`, `*wrap*`]`	additionally applies *wrap* to the collected numerators
`TagCollect[`*expr*`, `*tag*`, `*wrap*`]`	apply *wrap* to the part of *expr* tagged by *tag*
`TermCollect[`*expr*`]`	combines terms with common factors, i.e. does $ab + ac + d \to a(b + c) + d$
`Pool[`*expr*`]`	same as `TermCollect` but different method
`Pool[`*expr*`, `*wrap*`]`	additionally applies *wrap* to the $b + c$ part
`DotSimplify[`f_1`, `f_2`][`*expr*`]`	simplify *expr* with f_1 if free of any `NoBracket` items, else with f_2
`OnSize[`n_1`, `f_1`, `n_2`, `f_2`, ... , `$f_{\texttt{def}}$`][`*expr*`]`	returns f_1`[`*expr*`]` if `LeafCount[`*expr*`] <` n_1, f_2`[`*expr*`]` if `LeafCount[`*expr*`] <` n_2, etc., and $f_{\texttt{def}}$`[`*expr*`]` if the expression is still larger
`ExprHeads[`*expr*`]`	return all non-system symbols and heads in *expr*
`ExprParts[`*expr*`, `H`, `h`]`	return all subexpressions of *expr* which must contain the symbols and functions in H and may contain those in h
`Creep[`f`, `*patt*`...][`*expr*`]`	applies f to subexpressions of *expr* which contain only the patterns in *patt*
`MapOnly[`f`, `h`, `*patt*`...][`*expr*`]`	maps f onto subexpressions of head h in *expr* which must contain each of *patt* and no other symbols
`ApplyUnitarity[`*expr*`, `U`, `d`, `s`]`	simplify *expr* by exploiting the unitarity of the d-dimensional matrix U. The optional argument s specifies the simplification function to use internally (default: `FullSimplify`)
`ExpandSums[`*expr*`]`	turns all pieces of *expr* multiplied with `SumOver` into an actual sum
`SplitTerms[`f`, `*expr*`, `n`]`	applies f to *expr*, n terms at a time

`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 $ab + ac + d \to a(b + c) + d$ 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]
```

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

`ExprHeads` and `ExprParts` can be used to determine which parts of an expression are eligible e.g. for simplification. `ExprHeads` returns all non-system symbols and heads of an expression, while `ExprParts` picks subexpressions in which only selected heads appear.

`Creep` and `MapOnly` have similar functionality: they deliver a 'payload' function to parts of an expression containing certain objects only. `Creep` 'creeps' into an expression and applies its function as soon as a subexpression contains only the given patterns. `MapOnly` is more restrictive about the subexpression: For example, `MapOnly[`f`,` h`,` a`|`b`,` c`][`*expr*`]` applies f to subexpressions of head h which must contain c and a or b, and may not contain symbols other than a, b, c. Making c optional, `MapOnly[`f`,` h`,` a`|`b`, _|`c`][`*expr*`]` maps f onto subexpressions which must contain a or b and must not contain variables other that a, b, c.

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

`ExpandSums` turns expressions where index summations are denoted by `SumOver[`i`,` r`]` multiplied with the term to be summed over, such as appear e.g. in the output of `CalcFeynAmp`, into ordinary Mathematica sums. `ExpandSums[`*expr*`,` h`]` uses head h instead of `Plus`.

`SplitTerms` applies a function to a sum in batches, with at most the prescribed number of terms in each invocation. For all other heads the function is applied to the whole expression.

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. Code generation in C99 is available, too.

   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
`lumi_*.F`	luminosity computation
M`to`*N*`.F,.h`	the kinematics for a $M \to N$ process
`softphoton.F`	the integrals for the soft-photon approximation
`util.h`	definitions for the functions in `util.a`
`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 `util.a`
`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/models/`	code for model initialization:
`model_*.F,.h`	initialization of the stock models
`drivers/tools/`	helper scripts:
`mktm`	a script to set up Mathematica interfacing code (internal use)
`data`	a script for extracting the actual data out of the log files
`pnuglot`	a script for plotting data files using gnuplot
`sfx`	a script for making a self-extracting archive
`turnoff`	a script to turn on or off the evaluation of code modules
`submit`	a script to submit parallel jobs
`tools/`	tools for the numerical evaluation:
`ReadData.tm`	a program for reading data files into *Mathematica*
`reorder.c`	a utility to reorganize parameters in data files
`ScanGraphics.m`	special *Mathematica* graphics functions for parameter scans
`bfunc.m`	the explicit expressions for the one- and two-point functions
`btensor.m`	the explicit tensor reduction for the one- and two-point

functions in *Mathematica*

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`:

`SetupCodeDir[`*dir*`]`	create a directory *dir* 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.

Option	*default value*	
`Model`	`$Model`	the model used for the calculation, the presently initialized one by default
`Folder`	`models`	the subdirectory of the generated directory into which the model initialization code is written
`FileHeader`	`FileHeader`	the header for generated code files, same as for `WriteSquaredME` by default
`Drivers`	`"drivers"`	a directory containing customized versions of the driver programs which take precedence over the default ones in `$DriversDir`

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, `run.F`. 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 `run.F` in `$DriversDir`.

`SetupCodeDir` also sets up the model initialization, which consists of three files:

- `models/model.F` which contains the initialization routines of Sect. 5.1.3,
- `models/model.Fh`, model declarations for Fortran code,
- `models/model.ch`, model declarations for C code.

For *FeynArts*'s stock models the generated initialization merely uses/combines the ready-made files in `drivers/models`.

Models created with the *FeynArts* interface of *FeynRules* [Al09, Al14] come with a *model*`.pars` file containing numerical values for model parameters, and also the model file *model*`.mod` itself uses shorthands in coupling expressions, defined in the variable `FA$Couplings`.

Important: the numbers listed in the `.pars` file are usually tree-level values and may need to be adapted depending on the renormalization scheme, which is in the sole responsibility of the user.

The generation of initialization code is silently skipped if neither stock initialization files nor a `.pars` file is found on the `$ModelPath`.

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

option	*default value*	
`ExtraRules`	`{}`	extra rules to be applied to the expressions before the loop integrals are abbreviated
`TreeSquare`	`$TreeSquare` (default: `True`)	whether the square of the tree-level amplitude is added to the result
`LoopSquare`	`$LoopSquare` (default: `False`)	whether the square of the one-loop amplitude is added to the result
`Folder`	`"squaredme"`	the subdirectory of the code directory into which the generated code is written
`FilePrefix`	`""`	a string prepended to the filenames of the generated code
`SymbolPrefix`	`""`	a string prepended to global symbols to prevent collision of names when more than one process is linked

`FileHeader`	`"#if 0\n\`
	`* %f\n\`
	`* %d\n\`
	`* generated by FormCalc` *m.n* `%t\n\`
	`#endif\n\n"`
		the file header

`SubroutineIncludes`	`FileIncludes`	per-subroutine `#include` statements

`FileIncludes`	`{"#include \"decl.h\",`
	` "#include \"inline.h\"\n",`
	` "#include \"contains.h\"\n"}`
		per-file `#include` statements

`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 $2\,\mathrm{Re}\,\mathcal{M}^*_{\text{tree}}\mathcal{M}_{\text{1-loop}} + |\mathcal{M}_{\text{1-loop}}|^2$ rather than only $2\,\mathrm{Re}\,\mathcal{M}^*_{\text{tree}}\mathcal{M}_{\text{1-loop}}$. Without the $|\mathcal{M}_{\text{1-loop}}|^2$ term (which completes the square $|\mathcal{M}_{\text{tree}} + \mathcal{M}_{\text{1-loop}}|^2$), 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, $2\,\mathrm{Re}\,\mathcal{M}^*_{\text{tree}}\mathcal{M}_{\text{2-loop}}$, is much smaller than $|\mathcal{M}_{\text{1-loop}}|^2$ 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.

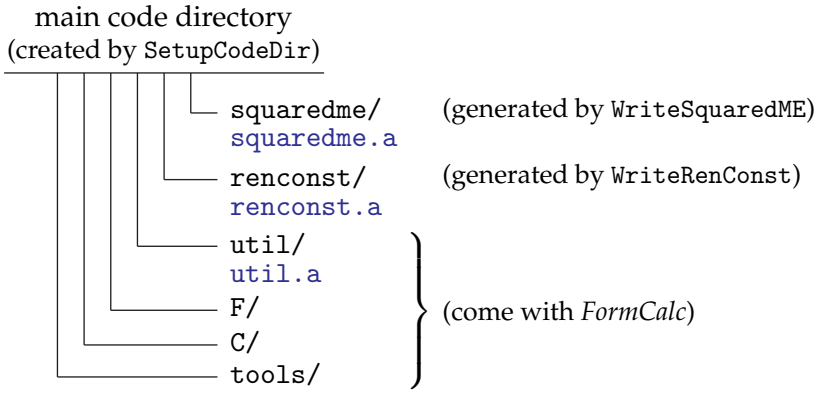
`SubroutineIncludes` gives declarations to be inserted in the declaration section of each subroutine, usually in the form of `#include` statements. In the most general form a list of three strings is given, the first of which included before, the second after local variable declarations, and the third 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.

`SetLanguage["Fortran"]`	set the output language to Fortran (default)
`SetLanguage["C"]`	set the output language to C99
`SetLanguage[`*lang*`, "novec"]`	generate unvectorized code

5.1.1 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.

5.1.2 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.

5.1.3 Specifying model parameters

Numerical model parameters are specified in the model initialization file. The initialization file must implement 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 center-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 α_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*AaA0i44*MH2)` instead of *Mathematica*'s default ordering `(3*Abb1*Alfa2*AaA0i44*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 vert
implicit none

#include "vars.h"
```

```
#include "inline.h"

        Cloop(HelInd(1)) = Cloop(HelInd(1)) +
     &    1/(4.D0*CW2**3*SW2**2)*
     &     ((8*MW2*(Pair1*Pair2*Sub437) - Alfa2*CW2**3*Sub445)/
     &        (MH2 - S) + (8*MW2*(Pair3*Pair4*Sub437) -
     &          Alfa2*CW2**3*Sub453)/(MH2 - T) +
     &       (8*MW2*(Pair5*Pair6*Sub437) - Alfa2*CW2**3*Sub461)/
     &        (MH2 - U))

#include "contains.h"

        end
```

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: `Pair`n (dot products), `Eps`n (contracted epsilon tensors), `F`n (fermion chains), `Aa`X`0i`n (loop integrals),
2. products or sums of variables of type 1: `Abb`n, `AbbSum`n,
3. matrix elements: `Mat`*Type*`(`i`,` j`)`.

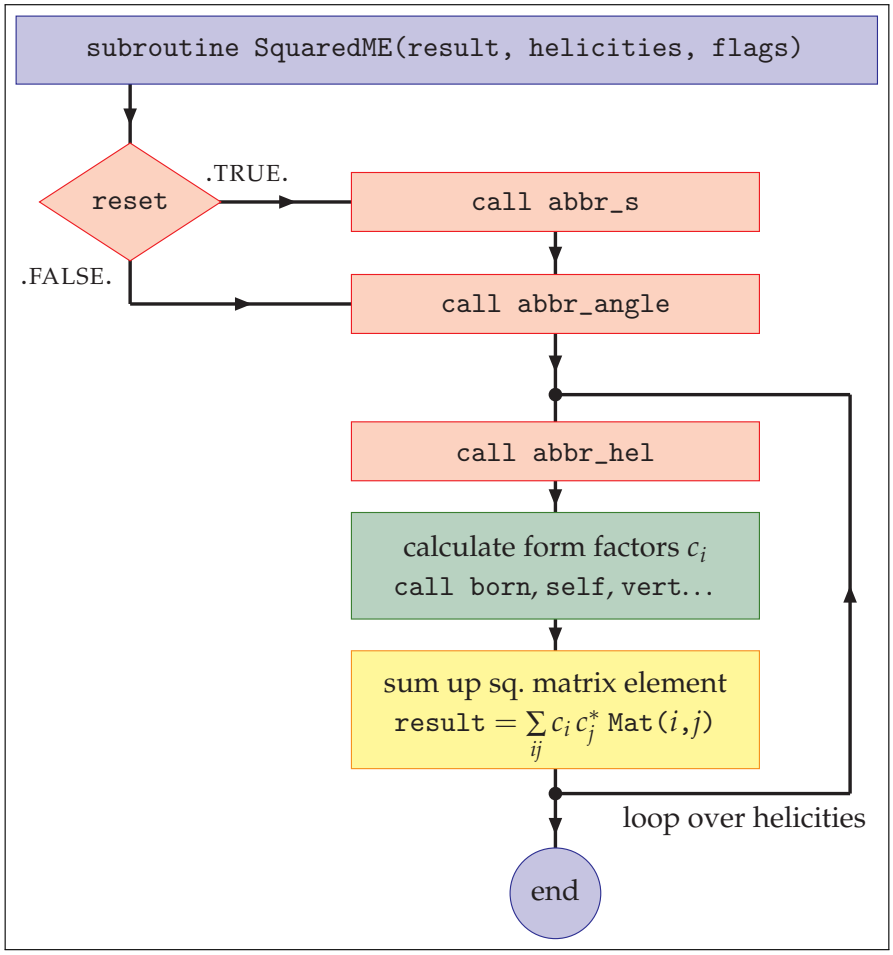
All of these abbreviations except the `Aa`X`0i`n are in fact the ones already introduced by *FormCalc*. These abbreviations share the general feature of being costly in CPU time (in particular the loop integrals), which means that for optimal performance they should be calculated as few times as possible. `WriteSquaredME` splits them into 2×3 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

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

Needless to say, each of these `abbri_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.



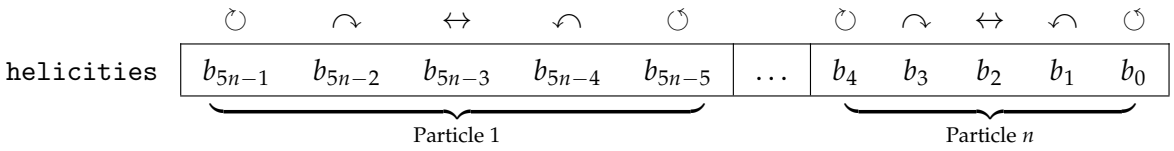
As long as one sticks to the default driver programs, one does not have to worry about actually calling `SquaredME`. For certain applications (e.g. Monte Carlo generators) it might nevertheless be attractive to invoke `SquaredME` directly. Its declaration is

```
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 `result(1)` $= |\mathcal{M}_{\text{tree}}|^2$ and `result(2)` $= 2\,\text{Re}\,\mathcal{M}^*_{\text{tree}}\mathcal{M}_{\text{1-loop}}$, except when there is no tree-level contribution, in which case `result(1)` $= 0$ and `result(2)` $= |\mathcal{M}_{\text{1-loop}}|^2$ 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:

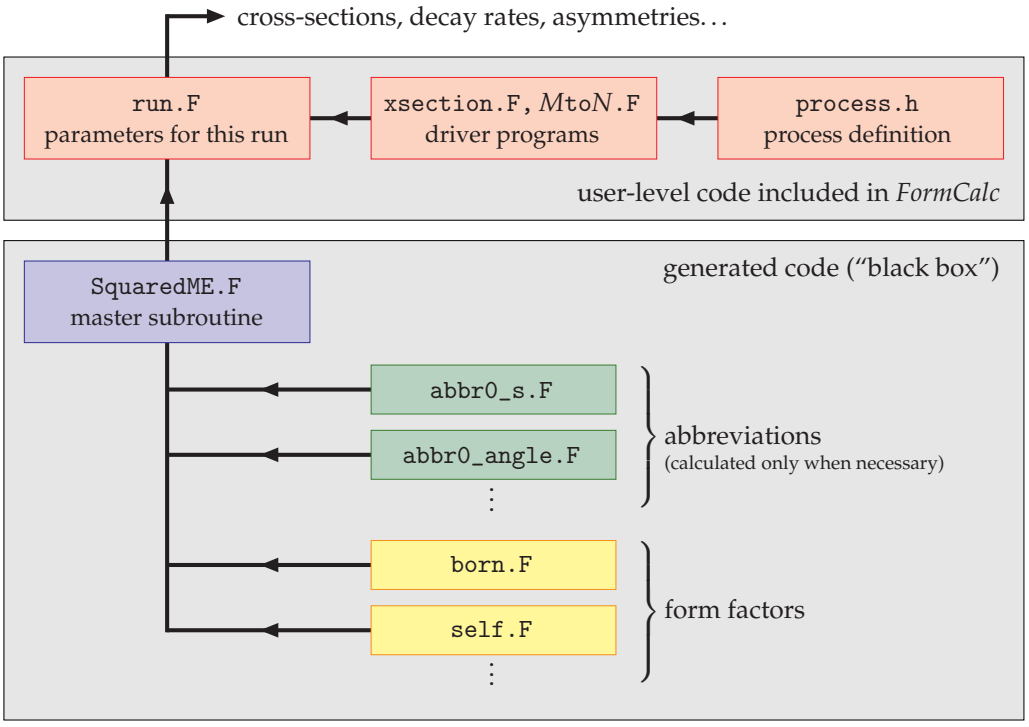


For instance, a left- and right-circularly polarized fermion scattering into an unpolarized two-vector-boson final state would be given by `helicities` $= 00010\,01000\,01110\,01110_2 = 74190_{10}$. Some compilers allow binary notation directly as `B'00010 01000 01110 01110'` (this is non-portable, however).

Currently three flags are passed:

- Bit 0 is the 'set mass' flag: if 1, the masses of the external particles are returned in the '`result`' argument. No cross-section is computed in this case.
- Bit 1 is the 'reset' flag: if 1, the abbreviations must be re-calculated completely. The latter is the case when the center-of-mass energy or the model parameters have changed.
- Bit 2 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

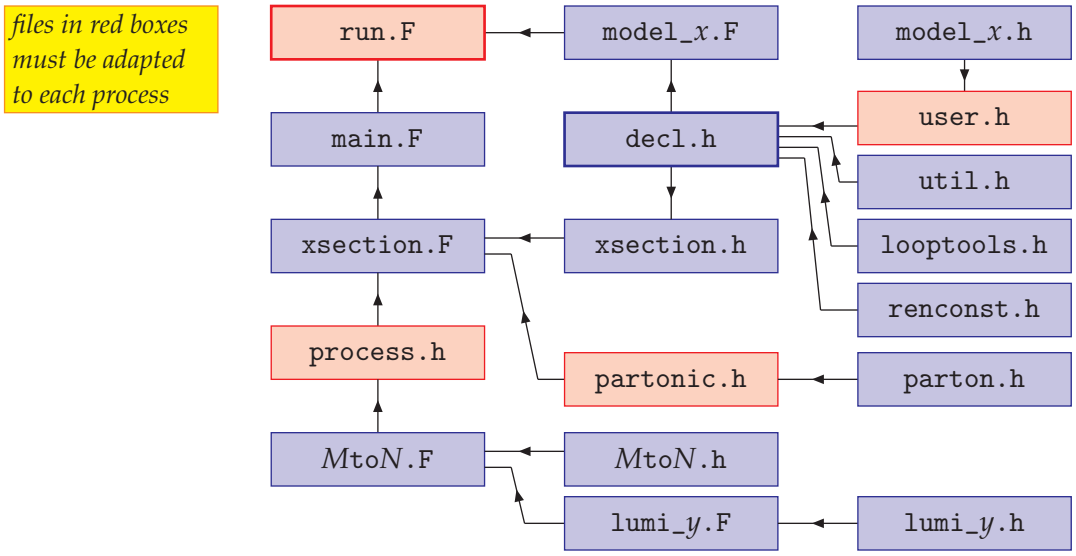
The code produced by `WriteSquaredME` needs in addition a driver program which supplies it with the necessary parameters, kinematics, etc. The default driver program that comes with *FormCalc* is organized in a very modular way and spreads out over several files: Technically, the main file is `run.F`, 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.' The actual main program, `main.F`, 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 relatively 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,
- *M*`toN.F` contains the kinematics-dependent code for a $M \to N$ process,
  M`toN.h` contains the declarations for *M*`toN.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`.

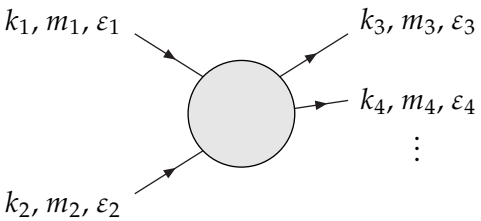
5.2.1 Process definition

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.

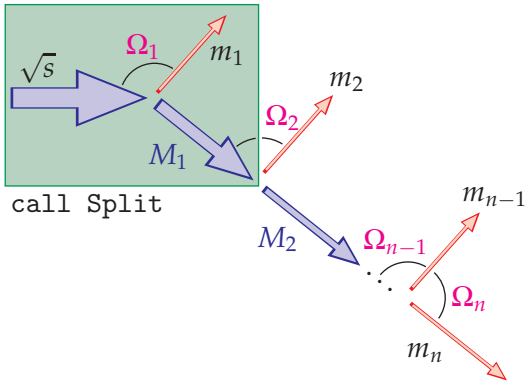


In addition to the external particles, the following items are defined in `process.h`:

- 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 bremsstrahlung shall be added (`PHOTONRADIATION`), 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 Ω_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 Ω-integrations. The corresponding phase-space parameterization is

$$\begin{aligned}
&\frac{1}{2\sqrt{s}} \int_{m_2+\cdots+m_n}^{\sqrt{s}-m_1} \mathrm{d}M_1\, \mathrm{d}\Omega_1\, \frac{k_1}{2} \\
&\times \int_{m_3+\cdots+m_n}^{M_1-m_2} \mathrm{d}M_2\, \mathrm{d}\Omega_2\, \frac{k_2}{2} \\
&\times \cdots \\
&\times \int_{m_n}^{M_{n-2}-m_{n-1}} \mathrm{d}M_{n-1}\, \mathrm{d}\Omega_{n-1}\, \frac{k_{n-1}}{2} \\
&\times \int \mathrm{d}\Omega_n\, \frac{k_n}{2}
\end{aligned}$$

where $\mathrm{d}\Omega_i = \mathrm{d}\cos\theta_i\, \mathrm{d}\varphi_i$. The particle's momentum k_i and $\cos\theta_i$ are given in the respective decay's rest frame. The φ_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 = mass(3) + mass(4) + mass(5)

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

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

  call VecSet(3, mass(3), 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 unambigously, 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.

After invoking `Split` or `VecSet` for external particle i, several kinematical quantities are available:

- `momspec(SPEC_M,`i`)` — mass m_i,
- `momspec(SPEC_E,`i`)` — energy E_i,
- `momspec(SPEC_K,`i`)` — momentum $|\vec{k}_i|$,
- `momspec(SPEC_ET,`i`)` — transverse energy E_i^T,
- `momspec(SPEC_KT,`i`)` — transverse momentum $|\vec{k}_i^T|$,
- `momspec(SPEC_RAP,`i`)` — rapidity y_i,
- `momspec(SPEC_PRAP,`i`)` — pseudo-rapidity η_i,

- `momspec(SPEC_DELTAK,`i`)` — the difference $E_i - k_i$,
- `momspec(SPEC_PHI,`i`)` — aximuthal angle φ_i,
- `momspec(SPEC_EX,`i`)`, `momspec(SPEC_EY,`i`)`, `momspec(SPEC_EZ,`i`)` — direction of motion $\vec{e}_i$.

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.
- `CutMin(`i`)`, `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

There are two principal ways to apply cuts in FormCalc. The first is by actually restricting the integration limits. The second is by selectively setting the integrand (differential cross-section) to zero whenever the cut condition(s) are met.

Restricting integration limits The subroutine `Split` allows to restrict the integration region of the M- and $\cos\theta$-integration. The φ-integration is not modified in the present setup. The application of cuts works e.g. 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, Re(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 'registered' by setting a bit in the integer variable `key` passed to `Split`, 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`). Available restrictions are:

Cuts restricting M_i

Cut on	Key
M_i	`CUT_MREM`
E_i	`CUT_MREM_E`
k_i	`CUT_MREM_K`
$E_{T,i}$	`CUT_MREM_ET`
$k_{T,i}$	`CUT_MREM_KT`
y_i	`CUT_MREM_RAP`
η_i	`CUT_MREM_PRAP`

Cuts restricting $\cos\theta_i$

Cut on	Key
$\cos\theta_i$	`CUT_COSTH`

The transverse energy cut has the slight anomaly that it corresponds to $E_T = \sqrt{k_T^2 + m^2}$ rather than $k^0\sqrt{e_x^2 + e_y^2}$ as the veto cut (see below). The reason is that, because the cuts are applied by actually restricting the integration bounds, solvability of the cut equations is a limiting factor.

Imposing veto cuts In many interesting cases, the cut condition(s) cannot straightforwardly be translated into restrictions of the integration limits. They are more easily applied through veto functions (1 in wanted, 0 in unwanted areas) cuts, i.e. by setting the integrand to zero in phase-space regions where cut conditions apply.

The veto function is constructed from the following preprocessor macros:

- `CUT_E(`i`)` — the energy E_i of particle i,

- `CUT_k(`i`)` — the momentum $|\vec{k}_i|$ of particle i,
- `CUT_ET(`i`)` — the transverse energy $E_i^T = k_i^0\sqrt{e_{ix}^2 + e_{iy}^2}$ of particle i,
- `CUT_kT(`i`)` — the transverse momentum $|\vec{k}_i^T|$ of particle i,
- `CUT_y(`i`)` — the rapidity y_i of particle i,
- `CUT_eta(`i`)` — the pseudo-rapidity η_i of particle i,
- `CUT_deltatheta(`i`)` — the scattering angle θ_i between particle i and the z-axis ($\leqslant \frac{\pi}{2}$), `CUT_cosdeltatheta(`i`)` — $\cos\theta_i$,
- `CUT_deltaalpha(`i`,`j`)` — the angle α_{ij} between particles i and j ($\leqslant \frac{\pi}{2}$), `CUT_cosdeltaalpha(`i`,`j`)` — $\cos\alpha_{ij}$,
- `CUT_deltay(`i`,`j`)` — the rapidity gap Δy_{ij} between particles i and j,
- `CUT_deltaeta(`i`,`j`)` — the pseudo-rapidity gap $\Delta\eta_{ij}$ between particles i and j,
- `CUT_R(`i`,`j`)` — the separation variable $\Delta R = \sqrt{\Delta y_{ij}^2 + \Delta\varphi_{ij}^2}$ of particles i and j,
- `CUT_rho(`i`,`j`)` — the separation variable $\Delta\rho = \sqrt{\Delta\eta_{ij}^2 + \Delta\varphi_{ij}^2}$ of particles i and j,
- `CUT_yprod(`i`,`j`)` — the opposite-hemisphere variable $y_i y_j$ of particles i and j,
- `CUT_etaprod(`i`,`j`)` — the opposite-hemisphere variable $\eta_i\eta_j$ of particles i and j,
- `CUT_invmass(`i`,`j`)` — the invariant mass M_{ij} of particles i and j.

The cuts are listed in the `CUT1...CUT20` definitions in `run.F`. Together they make up a single logical expression in Fortran and may include e.g. logical operators. For example, the following cut forces particle 3 to have a transverse energy of at least 10 GeV:

```
#define CUT1 CUT_ET(3) .lt. 10
```

5.2.5 Convolution

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

- `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
#define STATEFILE ""
#define SPIN -1

* for Vegas:
#define NSTART 1000
#define NINCREASE 500
#define NBATCH 1000
#define GRIDNO 0

* for Suave:
#define NNEW 1000
#define NMIN 2
#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 integra-

tors 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' 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.

`run` $p_1 \ldots p_n$ *sqrtS* [*serialfrom*[,*serialto*[,*serialstep*]]]
: compute the differential cross-section at a center-of-mass energy *sqrtS* with the external particles polarized according to $p_1 \ldots p_n$.

`run` $p_1 \ldots p_n$ *sqrtSfrom*,*sqrtSto*[,*sqrtSstep*] [*serialfrom*[,*serialto*[,*serialstep*]]]
: compute the integrated cross-section in the energy range *sqrtSfrom–sqrtSto* in steps of *sqrtSstep*, $p_1 \ldots p_n$ 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
- `l` for longitudinal polarization.

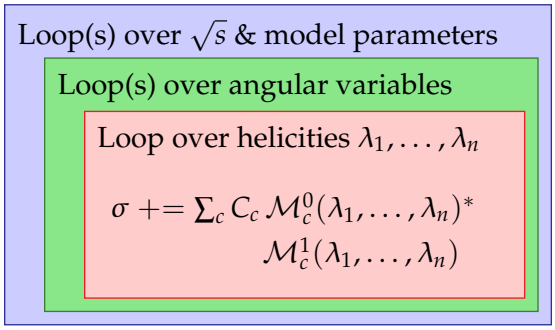
For a scalar particle, the polarization parameter must be present on the command line although its value is ignored. The energies are specified in GeV. If an energy is below threshold, `run` issues a warning and sets the energy to the threshold + 0.01 GeV.

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 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 vectorize the helicity loop using the CPU's SIMD instructions (that is, the squared matrix element is computed for several helicity combinations at once). 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.

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 $\left(\texttt{FCHSELEPS} \cdot \max_h t(h)\right)$ are actually computed. Unless one intentionally begins sampling e.g. on the borders of phase-space, the default values `FCHSELN` = 10 and `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...LOOP20`, 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 TB = 10, 50, 5
#define LOOP2 do 1 MA0 = 250, 1000, 250
```

result in a scan of the form

```
      do 1 TB = 10, 50, 5
      do 1 MA0 = 250, 1000, 250
      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 TB = 30
```

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 `SHOW` commands in the preprocessor variables `PRINT1...PRINT20` in `run.F`. The `SHOW` command two arguments, a string identifying the variable and then the variable itself. Defining, for example,

```
#define PRINT1 SHOW "Mh0=", Mh0
```

causes each data set to be preceded by a line of the form

```
# Mh0=  125.00
```

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 `ruptime` 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, 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 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`.

`pnuglot [`*opts*`]` *file*$_1$ *file*$_2$ ...	make a plot of the data files *file*$_1$, *file*$_2$, ...
options:	
`-o` *outfile*	how to name the output file: the plotting script will be called *outfile*`.gpl` and the actual plot *outfile*`.eps`. The default is *outfile* = *file*$_1$.
`-2`	use only columns 1:2 (usually the tree-level cross-section) for plotting
`-3`	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 β and `MA0` = M_{A^0}):

```
#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.

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

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*`,` arg_1`,` arg_2`, ...,` *options*`]`

`run[{`*sqrtSfrom*`,` *sqrtSto*`[,` *sqrtSstep*`]},` arg_1`,` arg_2`, ...,` *options*`]`

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

The arg_1, arg_2, ..., 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 distribute 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.

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

The parameters are stored in the form of rules in Mathematica, i.e. as *name* `->` *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 $2 \to 2$ 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.}] }
```

`DataRow[`v`,` r`,` e`]`	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 = {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 β 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 β 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.

`SelfEnergy[`i`->`f`,` m`]`	the self-energy $\Sigma_{if}(k^2 = m^2)$
`DSelfEnergy[`i`->`f`,` m`]`	the derivative $\partial\Sigma_{if}(k^2)/\partial k^2\big\vert_{k^2=m^2}$
`TreeCoupling[`i`->`f`]`	the tree-level contribution to $i \to f$ without external spinors
`VertexFunc[`i`->`f`]`	the one-loop contribution to $i \to f$ without external spinors
`LVectorCoeff[`*expr*`]`	the coefficient of $\not{k}P_L$ in *expr*
`RVectorCoeff[`*expr*`]`	the coefficient of $\not{k}P_R$ in *expr*
`LScalarCoeff[`*expr*`]`	the coefficient of P_L in *expr*
`RScalarCoeff[`*expr*`]`	the coefficient of P_R in *expr*
`ReTilde[`*expr*`]`	takes the real part of loop integrals in *expr*
`ImTilde[`*expr*`]`	takes the imaginary part of loop integrals in *expr*

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.)

`MassRC[`f`]`	the mass RC δM_f
`MassRC[`f_1`,` f_2`]`	the mass RC $\delta M_{f_1 f_2}$
`FieldRC[`f`]`	the field RC δZ_f
`FieldRC[`f_1`,` f_2`]`	the field RC $\delta Z_{f_1 f_2}$
`TadpoleRC[`f`]`	the tadpole RC δT_f
`WidthRC[`f`]`	the width Γ_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

$\partial\Sigma(k^2)/\partial k^2\big|_{k^2=m^2}$.

$$\delta M_f = \widetilde{\mathrm{Re}}\,\Sigma^F_{ff}(m_f, \tfrac{1}{2}, \tfrac{1}{2})\,, \qquad \delta M_B = \widetilde{\mathrm{Re}}\,\Sigma_{BB}(m_B^2)\,,$$

$$\delta M_{B_1B_2} = \frac{1}{2}\,\widetilde{\mathrm{Re}}\left(\Sigma_{B_2B_1}(m_{B_1}^2) + \Sigma_{B_2B_1}(m_{B_2}^2)\right),$$

$$\delta Z_f = -\,\widetilde{\mathrm{Re}}\begin{bmatrix}\Sigma^{VL}_{ff} + \partial\Sigma^F_{ff}(m_f, m_f, m_f)\\ \Sigma^{VR}_{ff} + \partial\Sigma^F_{ff}(m_f, m_f, m_f)\end{bmatrix}, \qquad \delta Z_B = -\,\widetilde{\mathrm{Re}}\,\partial\Sigma_{BB}(m_B^2)\,,$$

$$\delta Z_{f_1f_2} = \frac{2}{m_{f_1}^2 - m_{f_2}^2}\,\widetilde{\mathrm{Re}}\begin{bmatrix}\Sigma^F_{f_2f_1}(m_{f_2}, m_{f_2}, m_{f_1})\\ \Sigma^F_{f_2f_1}(m_{f_2}, m_{f_1}, m_{f_2})\end{bmatrix}, \quad \delta Z_{B_1B_2} = \frac{2}{m_{B_1}^2 - m_{B_2}^2}\,\widetilde{\mathrm{Re}}\,\Sigma_{B_2B_1}(m_{B_2}^2)\,,$$

$$\delta T_B = -\Sigma_B(m_B^2)\,,$$

$$\Gamma_f = \widetilde{\mathrm{Im}}\,\Sigma^F_{ff}(m_f, 1, 1)\,, \qquad \Gamma_B = \frac{1}{m_B}\,\widetilde{\mathrm{Im}}\,\Sigma_{BB}(m_B^2)\,,$$

where

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

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.

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

`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.

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

The calculation of the RCs and the underlying self-energies can be influenced and inspected in various ways.

The functions `SelfEnergy` and `DSelfEnergy`, used in the definitions of the RCs, are invoked when the RCs are calculated. They call `CreateTopologies`, `InsertFields`, `CreateFeynAmp`, and `CalcFeynAmp` and use whatever options have been set with `SetOptions` at that time. Computing the RCs with the same options as the virtual diagrams is just the right thing in most cases. For finer control, the amplitude generation can be modified in two ways.

Firstly, hooks are provided for `CreateTopologies`, `InsertFields`, and `CreateFeynAmp`.

`CreateTopologiesHook[`*args*`]`	`CreateTopologies` for `[D]SelfEnergy`
`InsertFieldsHook[`*args*`]`	`InsertFields` for `[D]SelfEnergy`
`CreateFeynAmpHook[`*args*`]`	`CreateFeynAmp` for `[D]SelfEnergy`

For example, `[D]SelfEnergy` does not call `InsertFields` directly, but the intermediate function `InsertFieldsHook` which by default simply redirects to the *FeynArts* function:

```
InsertFieldsHook[args__] := InsertFields[args]
```

`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.

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

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

Any CreateTopologies, InsertFields, and CreateFeynAmp options may be given here and apply only to the calculation of this particular renormalization constant.

The SEHook function finally governs setting the computed self-energy on-shell.

SEHook[*se*, *amp*, K2 -> m^2]	returns *amp* with K2 replaced by m^2

Attention: SEHook has attribute HoldAll (otherwise it would not be very useful). The *se* argument contains the original [D]SelfEnergy call and is for pattern matching or printout (in HoldForm) only – it will recurse if evaluated directly. The *amp* argument should not be evaluated more than once, as it triggers the computation of the self-energy. The m^2 argument should be evaluated after *amp* as it likely contains instances of TheMass which can successfully be resolved only after model initialization in *amp*.

The various intermediate results are stored in global variables where they can be inspected immediately after calling [D]SelfEnergy.

$RCTop	the output of CreateTopologies...
$RCIns	the output of InsertFields...
$RCAmp	the output of CreateFeynAmp...
$RCRes	the output of CalcFeynAmp...
	...in the last invocation of [D]SelfEnergy

During the calculation, PaintSE[$RCIns] and PutSE[{$RCTop,$RCIns,$RCAmp,$RCRes}] are called, which allows to paint the diagrams and store the intermediate results by setting $PaintSE and $PutSE, respectively.

PaintSE[*ins*]	same as PaintSE[*ins*, $PaintSE]
PaintSE[*ins*, True]	paint the diagrams *ins*
PaintSE[*ins*, *pre*]	paint the diagrams *ins* and store the output in *preN*.ps, where *N* = ProcName[*ins*]
PutSE[{*top*, *ins*, *amp*, *res*}]	same as PutSE[{*top*,*ins*,*amp*,*res*}, $PutSE]
PutSE[{*top*,*ins*,*amp*,*res*}, *pre*]	store the results in *preN*.{top,ins,amp,res}, where *N* = ProcName[*ins*]

variable	*default value*	
$PaintSE	False	whether to paint the diagrams
$PutSE	False	whether to save the intermediate results

If a string is given for $PaintSE or $PutSE it is used as a prefix for the filename and may

include a path name, where subdirectories are created as needed.

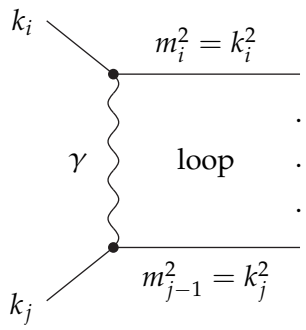
For performance reasons, `SelfEnergy` and `DSelfEnergy` remember the values they have computed. In unlikely cases it may happen that the user changes settings that affect the generation of diagrams but are not recognized by `SelfEnergy` and `DSelfEnergy`. This would mean that results are returned which 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:

`RCSub[`*subexpr*`]`	a function applied to subexpressions extracted by FORM
`RCInt[`*intcoeff*`]`	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 λ, 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 λ, but treats λ as an infinitesimal quantity, which means that terms of order λ or higher are discarded. In practice, λ 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 ra-

diation is always present experimentally: because photons are massless, they can have arbitrarily low energies and therefore escape undetected. So while the unphysical λ-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 α. Dealing with soft photons is relatively easy since the soft-photon cross-section is always proportional to the Born cross-section:

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

where "SB" stands for soft bremsstrahlung.

The soft-photon factor δ_{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 PHOTONRADIATION SOFT
#define ESOFTMAX .1D0*sqrtS
```

The first flag, `PHOTONRADIATION`, 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} }
```

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 `$`.

`ReadData[`f`,` n`,` h_{para}`,` h_{data}`]`	read the data file f into parameter and data sets in *Mathematica*; with the optional parameters n, h_{para}, and h_{data} one can choose to start numbering the sets with n, use head h_{para} instead of `Para`, and use head h_{data} instead of `Data`
`Data[`n`]`	the nth data set
`Para[`n`]`	the nth 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.)

function	*default value*	
`DataRow`	`List`	the function which is applied to each row of data in a file read by `ReadData`

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[§]; 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 und 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`.

`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 Δv_1 and Δv_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.

[§]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.

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

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

`MkDir["`*dir*$_1$`","`*dir*$_2$`",...]` makes sure the directory *dir*$_1$/*dir*$_2$/... exists, creating the individual subdirectories *dir*$_1$, *dir*$_2$, ... 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

`PrepareExpr[`*exprlist*`]`	prepare *exprlist* for write-out to code
`WriteExpr[`*file*`,` *exprlist*`]`	write *exprlist* to *file*
`CodeExpr[`*v*`,` *t*`,` *exprlist*`]`	the expressions *exprlist* with variables *v* and temporary variables *t* in a form ready for write-out by `WriteExpr`
`RuleAdd[`*a*`,`*b*`]`	same as *a* `->` *a* `+` *b*
`DebugLine[`*s*`]`	a debugging statement for variable *s*
`NoDebug[`*expr*`]`	do not generate a debugging statement for *expr*
`$SymbolPrefix`	a string prepended to all externally visible symbols to avoid symbol conflicts

`PrepareExpr[{`*var*$_1$ `->` *expr*$_1$`,` *var*$_2$ `->` *expr*$_2$`,...}]` 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*$_1$ `->` *expr*$_1$`,` *var*$_2$ `->` *expr*$_2$`, ...}`, 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* `->` *var* `+` *expr*.

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

Assignments of the form `NoDebug[`*var* `->` *expr*`]` never generate debugging/checking statements, regardless of the value of `DebugLines` or `DebugLabel`.

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

PrepareExpr option	*default value*	
`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 mininum leaf count above which a subexpression becomes eligible for abbreviationing
`DebugLines`	`0`	whether to generate debugging/ checking statements for every variable
`DebugLabel`	`True`	whether and which label to use in debugging/checking statements
`MakeTmp`	`Identity`	a function for introducing user-defined temporary variables
`Declarations`	`(Rule | RuleAdd)[`*v*`_, _] :>` *v*	a `Cases` pattern for selecting variable declarations
`FinalTouch`	`Identity`	a function which is applied to the final subexpressions
`ResetNumbering`	`True`	whether to number `dup`*n* and `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 and/or checking statements are generated for each expression. Admissible values are 0 = no statements are generated, 1 = debugging statements are generated, 2 = checking statements are generated, 3 = debugging and checking statements are generated. Debugging messages are usually generated for the expressions specified by the user only. To cover intermediate variables (e.g. the ones introduced for optimization), too, specify the negative of the values above.

`DebugLabel` specifies with which label debugging/checking statements are printed. `False` disables printing, `True` prints the variable name, and a string prefixes the variable name. Any other value is understood as a function which is queried for each variable assignment and its output, `True`, `False`, or a string, individually determines generation of the debug statement for each variable assignment.

Debugging/checking statements come in three kinds, 1, -2, and 2. Type 1 are debugging statements of the form

var `=` *expr*
`DEB("`*var*`",` *var*`)`

Type -2 and 2 are (pre and post) checking statements of the form

`CHK_PRE(`*var*`)`
var `=` *expr*
`CHK_POST("`*var*`",` *var*`)`

The actual statements are constructed from `$DebugCmd[`t`]`, $t = 1, -2, 2$, which gives the debugging statement in `StringForm` format with three arguments: prefix (string), variable name (string), variable value (number). For example, the default setting for debugging statements is

```
$DebugCmd[1] = "DEB(\"‘1‘‘2‘\", ‘3‘)\n"
```

which refers to the preprocessor macro `DEB` that might be defined in Fortran as

```
#define DEB(tag,var) print *, tag, var
```

The statements are further enclosed by the strings in `$DebugPre[`t`]` and `$DebugPost[`t`]`, which serves e.g. to enable debugging, as in:

```
$DebugPre[1, level_:4] := "#if DEBUG >= " <> ToString[level] <> "\n"
$DebugPost[1] = "#endif\n"
```

Here debugging would be activated at compile time if the preprocessor variable `DEBUG` (debug level) is larger than 2.

`MakeTmp` specifies a function for introducing user-defined temporary variables, e.g. `ToVars`.

`Declarations` specifies a pattern suitable for use in `Cases` that selects all objects to be declared as variables.

`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.

The prefixes for temporary and optimization variables, i.e. the `tmp` in `tmp123` and the `dup` in `dup123`, may respectively be changed with `$TmpPrefix` and `$DupPrefix`.

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

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

`DeclIf ->` *var*, where *var* is a string suitable for a preprocessor variable, separates declarations and code in the form

```
#ifndef var
#define var
[declarations]
#else
[code]
#endif
```

A file so generated is supposed to be included once in the declarations section and once in the code part. This can be necessary in particular in Fortran, if e.g. non-declaration statements such as statement functions need to be placed between declarations and code.

`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

`OnePassOrder[`*list*`]`	order *list* such that the definition of each item comes before its first use
`MoveDepsRight[`r_1`,...,`r_n`]`	move variables among the lists $r_1, \ldots, r_n$ such that a definition does not depend on r_i further to the right
`MoveDepsLeft[`r_1`,...,`r_n`]`	move variables among the lists $r_1, \ldots, r_n$ such that a definition does not depend on r_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[`r_1`,...,`r_n`]` 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 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[`r_1`, ..., `r_n`]` 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`.

`ToVars[`*patt*`,` *name*`][`*exprlist*`]`	introduce variables for all subexpressions in *exprlist* matching *patt*

`ToVars[`*patt*`,` *name*`][`*exprlist*`]` introduces variables for subexpressions matching *patt* and expects to be used on an expression list such as given to `PrepareExpr`. To make the new variables temporary (in the `CodeExpr` sense), `ToVars` should be applied through `PrepareExpr`'s `MakeTmp` option. Unlike with `Abbreviate`, the variables introduced are not subscripted by loop indices and hence also not hoisted outside a `DoLoop`. The new variable definitions are inserted at the right places into the *exprlist*.

Why move certain subexpressions to variables at all? Firstly, it makes it easier to inspect/debug their values, using the `DebugLine` option. Secondly, and in contrast to abbreviations introduced for optimization reasons (e.g. by the `Expensive` or `Optimize` options), it can improve the readability of the generated code by well-chosen variable names.

The names for the variables are determined by the function *name* which receives the expression being abbreviated and must return a symbol name for it. If *name* is a string instead, `NewSymbol[`*name*`, 0]` is taken as naming function, i.e. the variable names will be *name*`1`, *name*`2`, etc. The numbering is consecutive across `ToVars` calls but can be reset by assigning `SymbolNumber[`*name*`] = 0`.

For example, `PrepareExpr[`*exprlist*`, MakeTmp -> ToVars[LoopIntegral[__], Head]]` introduces variables for all loop integrals in *exprlist*, with names like `B0i1`, `B0i2`, etc.

`PaVeIntegral`	a pattern matching the head of all one-loop Passarino–Veltman integrals (`A0i`, `B0i`, etc.)
`CutIntegral`	a pattern matching the head of all one-loop OPP integrals (`Acut`, `Bcut`, etc.)
`LoopIntegral`	the union of `PaVeIntegral` and `CutIntegral`

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

`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*.

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

`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[`$cond_1$`,` a_1`,` $cond_2$`,` a_2`, ...]`, which is equivalent to `IndexIf[`$cond_1$`,` a_1`, IndexIf[`$cond_2$`,` a_2`, ...]]`. 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 δ_{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`.

`BlockSplit[`*var* `->` *expr*`]`	split *expr* into subexpressions with leaf count less than `$BlockSize`
`FileSplit[`*exprlist*`,` *mod*`,` *writemod*`,` *writeall*`]`	split *exprlist* into batches with leaf count less than `$FileSize`, call *writemod* to write a each module to file and finally *writeall* to generate a 'master subroutine' which invokes the modules
`ToArray[`*s*`]`	take the symbol *s* apart into letters and digits, e.g. `X123` → `X[123]`
`Renumber[`*expr*`,` v_1`,` v_2`, ...]`	renumber all $v_1[n]$, $v_2[n]$, ... in *expr*
`MaxDims[`*args*`]`	find the maximum indices of all functions in *args*
`$BlockSize`	the size of each block for `BlockSplit`
`$FileSize`	the size of each file for `FileSplit`

`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*`,` s_1`,` s_2`,...]` turns all occurrences of the symbols s_1NNN, s_2NNN, etc. in *expr* into $s_1[NNN]$, $s_2[NNN]$, etc.

`Renumber[`*expr*`,` var_1`,` var_2`, ...]` renumbers all $var_1[n]$, $var_2[n]$, ... 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[`*name*`]`	the declaration for the subroutine *name*
`VarDecl[`$\underbrace{\textit{vars}, \textit{type}}_{a}$`]`	the declaration of *vars* as variables of type *type*
`VarDecl[Common[`*com*`][`*a*`]]`	the declaration of common block *com* with individual variables *a* (*a* as above)
`VarDecl[NameMap[`*com*`][`*a*`]]`	the declaration of common block *com* with an array plus a preprocessor map of the *a* onto array elements (*a* as above)
`VarDecl[NotEmpty[`*a*`]]`	output only if at least one variable list in *a* is non-empty
`DoDecl[`*i*`,` *range*`]`	the declaration of a do-loop of *i* over *range*
`CallDecl[`*subs*`]`	the invocations of subroutines *subs*
`Dim[`*i*`]`	the highest value the index *i* takes on
`DoDim[`*i*`]`	like `Dim[`*i*`]` but including indices from `SumOver` in the amplitudes
`Enum[`*names*`]`	set up *names* as named indices
`ClearEnum[]`	clear `Enum` definitions

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

`VarDecl[{`$v_1, v_2, \ldots$`},` *t*`]` returns a string with the declaration of $v_1, v_2, \ldots$ as variables of type *t*. Any other strings are output verbatim, e.g. `"#ifdef COND\n"`, `"#endif\n"`.

`VarDecl[Common[`*c*`][{`$v_1, v_2, \ldots$`},` *t*`]]` declares $v_1, v_2, \ldots$ to be members of common block *c*.

`VarDecl[NameMap[`*c*`][{`$v_1, v_2, \ldots$`},` *t*`]]` works much like `Common` but puts only arrays into the common block (one for each data type), together with preprocessor statements mapping the $v_1, v_2, \ldots$ 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 containing 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[{`$sub_1, sub_2, \ldots$`}]` returns a string with the invocations of the subroutines sub_1, sub_2, ..., taking into account possible loops indicated by `DoLoop`.

`Dim[`*i*`]` and `DoDim[`*i*`]` return the highest value the index *i* takes on, excluding (`Dim`) or including (`DoDim`) indices found in `SumOver` statements of amplitudes evaluated so far. A manual

assignment `Dim[`i`] =` n generates correct array dimensions for index i only. A manual assignment `DoDim[`i`] =` n also generates a loop over index i.

`Enum` associates index names with integers, used for determining array bounds during code generation (the index names themselves remain unchanged). The syntax is similar to that of C's `enum`, e.g. `Enum[`a`,` b`,` c `->5,` d`]` associates $a \to 1$, $b \to 2$, $c \to 5$, $d \to 6$. `ClearEnum[]` removes any `Enum` assignments previously made.

7.5 Compatibility Functions

`ToOldBRules`	rules to convert to the conventions for two-point functions of *LoopTools* 2.2 and before
`ToNewBRules`	rules to convert to the conventions for two-point functions of *LoopTools* 2.3 or later

`ToOldBRules` and `ToNewBRules` are two sets of transformation rules that convert between the old (`B0`, `B1`, ...) and new (`B0i[b0,b1,...]`) conventions for two-point functions in *LoopTools*.

References

[Al09] *Comp. Phys. Commun.* **180** (2009) 1614 [arXiv:0806.4194].

[Al14] A. Alloul, N. Christensen, C. Degrande, C. Duhr, B. Fuks *Comp. Phys. Commun.* **185** (2014) 2250 [arXiv:1310.1921].

[Ch79] M. Chanowitz, M. Furman, I. Hinchliffe, *Nucl. Phys.* **B159** (1979) 225.

[Cv76] P. Cvitanovic, *Phys. Rev.* **D14** (1976) 1536.

[dA98] F. del Aguila, A. Culatti, R. Muñoz Tapia, 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, 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 Schladming 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