FormCalc and *LoopTools* 1.2 User's Guide

July 31, 1998 Thomas Hahn

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

1) We don't promise that this software works. (But if you find any bugs, please let us know!)

2) You can use this software for whatever you want. You don't have to pay us.

3) You may not pretend that you wrote this software. If you use it in a program, you must acknowledge somewhere in your documentation that you've used our code.

If you're a lawyer, you will rejoice at the exact wording of the license at `http://www.fsf.org/copyleft/lgpl.html`.

FormCalc is available from `http://www-itp.physik.uni-karlsruhe.de/formcalc`.

LoopTools is available from `http://www-itp.physik.uni-karlsruhe.de/looptools`.

FeynArts is available from `ftp://ftp.physik.uni-wuerzburg.de/pub/hep/index.html`.

FF is available from `ftp://ftp.nikhef.nl/pub/ff/FF.html`.

If you make this software available to others please provide them with this manual also. There exists a mailing list for each program where updates will be announced. To subscribe to these lists, send a mail (any text) to `hahn-formcalc-subscribe` or `hahn-looptools-subscribe@particle.physik.uni-karlsruhe.de`, respectively.

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

Thomas Hahn
Institut für Theoretische Physik
Universität Karlsruhe
e-mail: `hahn@particle.physik.uni-karlsruhe.de`

Contents

1 *FormCalc* 5

1.1 General Considerations . . . 5
1.2 Getting Started . . . 7
1.3 Running *FormCalc* . . . 8
1.3.1 Generating the Diagrams . . . 8
1.3.2 Evaluating Diagrams . . . 8
1.3.3 Processing Entire Files . . . 15
1.3.4 Handling Abbreviations . . . 16
1.4 A Demonstration Run . . . 19

2 *LoopTools* 23

2.1 Making the Libraries . . . 23
2.2 One-Loop Integrals . . . 25
2.2.1 Tensor Coefficients . . . 26
2.2.2 Conventions for the Momenta . . . 27
2.3 Functions provided by *LoopTools* . . . 28
2.3.1 One-point function . . . 28
2.3.2 Two-point functions . . . 28
2.3.3 Derivatives of Two-point functions . . . 28
2.3.4 Three-point functions . . . 29
2.3.5 Four-point functions . . . 30
2.3.6 On Warning Messages . . . 31
2.4 Using *LoopTools* with Fortran . . . 31
2.5 Using *LoopTools* with C++ . . . 35
2.6 Using *LoopTools* with *Mathematica* . . . 38
2.7 Remarks on Quadruple Precision . . . 39

A The original *FF* Manual **43**

A.1 Introduction . . . 43

A.2 Brief description of the scalar loop routines . . . 44

A.2.1 Overview . . . 44

A.2.2 Timings . . . 46

A.2.3 Tests . . . 47

A.3 Installation . . . 47

A.4 Initialisation . . . 49

A.5 The error reporting system . . . 50

A.5.1 Overview . . . 50

A.5.2 Using the system . . . 50

A.5.3 Debugging possibilities . . . 52

A.5.4 Summary . . . 53

A.6 Scalar n-point functions . . . 53

A.6.1 One-point function . . . 54

A.6.2 Two-point function . . . 54

A.6.3 Three-point function . . . 55

A.6.4 Four-point function . . . 56

A.6.5 Five-point function . . . 58

A.7 Tensor integrals . . . 59

A.7.1 Vector integrals . . . 59

A.8 Determinants . . . 60

A.8.1 2×2 determinants . . . 61

A.8.2 3×3 determinants . . . 62

A.8.3 4×4 determinants . . . 63

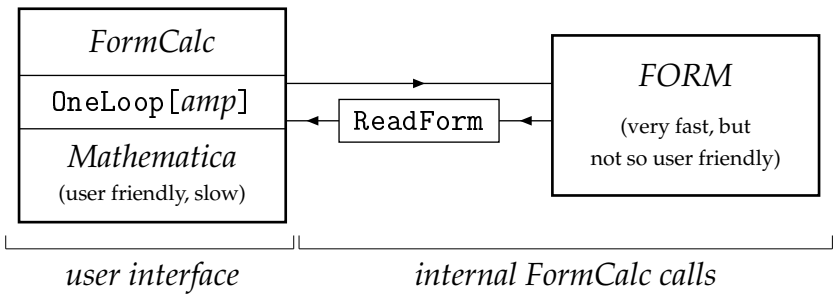
1 *FormCalc*

1.1 General Considerations

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

FormCalc calculates and simplifies one-loop Feynman diagrams. It reads amplitudes generated with *FeynArts* [KüBD91] and returns the results in a way well suited for further numerical (or analytical) evaluation.

Internally, *FormCalc* delegates the hard work (e.g. working out fermionic traces) to *FORM* by Jos Vermaseren [Ve91]. Thus *FormCalc* is merely a driver that threads the *FeynArts* amplitudes through *FORM* in an appropriate way. The idea is rather simple: *FormCalc* prepares the symbolic expressions of the diagrams in an input file for *FORM*, runs *FORM*, and retrieves the results. This interaction is transparent to the user. 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 about 10 minutes on an ordinary Pentium PC.

One important aspect of *FormCalc* is that it automatically gathers spinor chains, scalar products of and antisymmetric tensors contracted with momenta and/or polarization 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 the standard di-

mensional regularization, in 4 dimensions it uses the method of constrained differential renormalization which at one-loop level is equivalent to dimensional reduction. Details on these methods can be found in [HaP98].

A one-loop calculation generally includes three steps:

Step	Program
1. Generate the diagrams	*FeynArts*
2. Simplify analytically	*FormCalc*
3. Evaluate numerically	(*LoopTools*)

FormCalc does not fully automatize one-loop calculations to the extent that it directly produces cross-sections from the process specification. The reason is that at one-loop level already many features appear which must be treated differently from process to process. Nevertheless, the *FormCalc* output can easily be evaluated further, e.g. numerically in a Fortran program. One big advantage is that *FormCalc* returns a *Mathematica* expression which is considerably easier to modify than Fortran code.

The numerical evaluation finally involves writing own programs which are fine-tuned to the requirements of a specific process. This is aided by the *LoopTools* package which provides all functions needed for code produced from *FormCalc* output. Typically, two steps are involved: a) write a program to convert the *Mathematica* expressions into Fortran code, b) write a driver program to run that code. To demonstrate how this can be done, a full non-trivial one-loop calculation—elastic Z–Z scattering in the electroweak Standard Model ($\approx$ 500 diagrams [DeDH97])—is included in the *FormCalc* distribution.

1.2 Getting Started

To run *FormCalc* you need *Mathematica* 2.2 or above, *FORM* 2 or above* and `gcc`, the GNU C compiler. *FormCalc* comes in a compressed tar archive `FormCalc12.tar.gz`. To install it, create a directory for *FormCalc*—`FC12`, say—and unpack the archive there:

```
mkdir FC12
cd FC12
gunzip -c ../FormCalc12.tar.gz | tar xvf -
./compile
```

The last line compiles the `ReadForm` program which reads and translates the *FORM* output into *Mathematica*. The `compile` script creates binaries whose name has the system type appended, so the binaries for different platforms can coexist in one directory.

The directory structure after installing *FormCalc* should be

`FC12/`	the *FormCalc* directory
`FC12/FormCalc.m`	the main program
`FC12/form.set`	setup parameters for *FORM*
`FC12/pave.frm`	the tensor reduction code
`FC12/ReadForm.tm`	`ReadForm` source code
`FC12/ReadForm_`*sysid*	`ReadForm` executable
`FC12/compile`	shell script to compile `ReadForm.tm`
`FC12/README`	additional information about this release
`FC12/demo`	directory containing demo programs

sysid is a string that identifies the platform; it is the value of the environment variable `HOSTTYPE` in the Bourne shell (`sh`) and typically has values such as `alpha`, `i486`, etc.

If *FORM* is invoked on your system not just by `form`, edit `FormCalc.m`, locate the line

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

and insert there the right command for running *FORM* (`Form`, `form2`, ...).

FormCalc runs also with *FORM* 1, but cannot fully simplify spinor chains with external fermions then.

1.3 Running *FormCalc*

1.3.1 Generating the Diagrams

FormCalc calculates diagrams generated by *FeynArts*. In principle, the diagrams could also be entered by hand using the *FeynArts* conventions, but this is rather cumbersome. *FormCalc* follows the conventions of *FeynArts* 1. Higher versions of *FeynArts* use a more elaborate output format but allow to translate the results back to the old conventions, e.g. in the current version 2.2 the function `ToFA1Conventions` is supplied.

Usually it is 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.

Important: It is *not* possible to load *FeynArts* and *FormCalc* into one *Mathematica* session.

1.3.2 Evaluating Diagrams

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

`OneLoop[`*amp*`]`	simplify the amplitudes *amp*
`Pick[`*amp*`,` *sel*`]`	pick the diagrams specified with *sel* from *amp* *sel* can be e.g. `{5, {40, 57}, 79}` which selects diagrams 5, 79, and 40 through 57, or `All`
`FermionicQ[`*d*`]`	`True` if diagram *d* contains fermions
`DiagramType[`*d*`]`	returns the number of propagators in diagram *d* not belonging to the loop

The name `OneLoop` is not strictly correct because the function accepts also tree-level diagrams. It performs the following simplifications:

- indices are contracted as far as possible,
- fermion traces are evaluated,

- open fermion chains are simplified using the Dirac equation,
- the tensor reduction is performed,
- local terms are added[†],
- the results are partially factored,
- abbreviations are introduced.

`OneLoop` is used like this:

```
math
<< FormCalc.m;
amps = << myamps.m;
alldiags = OneLoop[amps]
```

The file `myamps.m` is assumed here to contain amplitudes generated by *FeynArts*. Alternatively, if one wants to evaluate only a subset of diagrams,

```
somediags = OneLoop[ Pick[amps, {3, 5, {21, 29}}] ]
```

will calculate diagrams 3, 5, and 21 through 29, or for a single diagram it can be just

```
onediag = OneLoop[ amps[[9]] ]
```

FormCalc offers two functions which are helpful to select diagrams according to certain criteria instead of picking them by numbers.

`FermionicQ` is useful for selecting diagrams that contain fermions, and is used like

```
fermamps = Select[amps, FermionicQ]
```

`DiagramType` returns the number of propagators not belonging to the loop. For example, in a $2 \to 2$ process without self-energy insertions on external legs[‡], `DiagramType` gives 2 for a self-energy diagram, 1 for a vertex-correction diagram, and 0 for a box diagram, so that for instance

[†]In D dimensions, the divergent integrals are expanded in $\varepsilon = D - 4$ 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 $g^{\mu}{}_{\mu}$'s). 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 [dACTP98].

[‡]This may seem a strange selection of diagrams, but is in fact just what one needs in an on-shell renormalization scheme where there is no external wave-function renormalization.

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

selects all vertex corrections.

It may be questioned how useful the indirect information `DiagramType` returns really is. However, topological information is contained in an amplitude only indirectly, too, so `DiagramType` is, in a way of speaking, a "morning-after" remedy. A safe and much more versatile way of selecting diagrams according to topological criteria is by using the `ExcludeTopologies` filters in *FeynArts*.

`OneLoop` needs no declarations of the kinematics of the underlying process; it uses the information *FeynArts* hands down. When amplitudes are loaded, *FormCalc* automatically introduces kinematical invariants for some common processes[§], e.g. Mandelstam variables for a $2 \to 2$ process.

`k[`n`]`	nth external momentum
`e[`n`]`	nth polarization vector
`S`	Mandelstam variable $s =$ `(k[1] + k[2])`^{2}
`T`	Mandelstam variable $t =$ `(k[1] − k[3])`^{2}
`U`	Mandelstam variable $u =$ `(k[1] − k[4])`^{2}
`Sf`	extended Mandelstam variable $s' =$ `(k[3] + k[4])`^{2} (used in a $2 \to 3$ reaction)
`Tf`	extended Mandelstam variable $t' =$ `(k[2] − k[4])`^{2} (used in a $2 \to 3$ reaction)
`Uf`	extended Mandelstam variable $u' =$ `(k[2] − k[3])`^{2} (used in a $2 \to 3$ reaction)
`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 $\varepsilon_{0123} = +1$
`ga[k[`n`]]`	γ_μ `k[`n`]`$^\mu$
`ga[li`n`]`	$\gamma_{\mathtt{li}n}$ (`li`n represents Lorentz index n)
`ga5`	γ_5

[§]Currently processes with 4 or 5 external legs, i.e. $2 \to 2$, $2 \to 3$, $1 \to 3$, $1 \to 4$.

The numbering of the momenta and polarization vectors is consecutive, i.e. the incoming momenta are numbered $1 \ldots n_{\text{in}}$ and the outgoing momenta $n_{\text{in}} + 1 \ldots n_{\text{in}} + n_{\text{out}}$.

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

The elements of a spinor chain are multiplied with `**` (`NonCommutativeMultiply`) to preserve their order.

The `OneLoop` results are subject to the setting of the following options.

option	*default value*	
`CancelEps`	`False`	whether to discard terms containing antisymmetric tensors early in the calculation
`ReduceME`	`Automatic`	whether to eliminate one external momentum via momentum conservation
`FormCommands`	`""`	extra *FORM* commands to be executed directly after simplifying each diagram but before the tensor reduction
`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*

`CancelEps` is for used for processes with a forward–backward symmetry. In such processes no antisymmetric tensors must appear. Normally, these terms do drop out in the end as they should, but possibly not in every part of the amplitude separately. Setting `CancelEps -> True` will discard terms containing the antisymmetric Levi-Civita tensor at an early stage. This can also speed up the calculation.

An amplitude from *FeynArts* in general depends on all external momenta. It turns out not to be economical for the size of the result to eliminate one external momentum altogether via momentum conservation. However, it is sometimes worthwhile doing so in dot products only, the main effect of which is that it reduces the number of abbreviations. This is what *FormCalc* does when `ReduceME -> True` is set. Empirically, the

savings are large for bosonic amplitudes but unconvincing for fermionic ones. The default setting `Automatic` is a trade-off between speed and the most compact result; it evaluates to `True` for bosonic and to `False` for fermionic amplitudes.

`FormCommands` is an option for *FORM* specialists with which extra *FORM* commands can be specified. They will be executed for each diagram after contracting the indices and performing the trace but before the tensor reduction. The syntax is `FormCommands -> "`*cmd1*`\n`*cmd2*`\n...".`

`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 `t20373909610282333891.frm`. `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`.

It is important to know that *FormCalc* sets up transformation rules in preparation for `OneLoop` the moment an amplitude is loaded, e.g. such that dot products of momenta are automatically simplified as for on-shell particles. This setup can be influenced by the following system variables. It should be clear from the aforesaid that some of these variables must be set *before* loading the amplitudes.

variable	*default value*	
`$OnShell` **(must be set *before* loading amplitudes!)**	`True`	whether to put the external particles on their mass shell
`$Transversality` **(must be set *before* loading amplitudes!)**	`True`	whether polarization vectors should be orthogonal to the corresponding momentum
`$Dimension`	`D`	dimension to calculate in (`4` or `D`)

`$OnShell` determines whether external particles are on their mass shell, i.e. it sets $p_i^2 = m_i^2$ where p_i is the ith external momentum and m_i the corresponding mass.

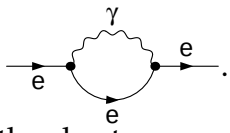
`$Transversality = True` enforces orthogonality of the polarization vectors of external vector bosons and their corresponding momentum, i.e. it sets $\varepsilon_{i,\mu} p_i^\mu = 0$ where p_i and ε_i are the ith external momentum and polarization vector, respectively.

`$Dimension` specifies in which space-time dimension 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 [HaP98].

- The default `$Dimension = D` corresponds to **dimensional regularization** [tHV72]. 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 [ChFH79] and works with an anticommuting γ_5 in all dimensions.[¶]

- Alternatively, `$Dimension = 4` selects the method of **constrained differential renormalization** (CDR) [dACTP98]. 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 [HaP98] 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.

The results of both schemes can differ by finite (local) terms. A simple example where this is the case is the electron self-energy in QED, 

. Assuming that its amplitude has already been created with *FeynArts*, the short program

```
<< FormCalc.m;
$OnShell = False;
$Dimension = D;
OneLoop[<< electronSE.amp] //. Abbreviations[]
```

[¶]In fact, *FormCalc* relies on the well-tested algorithms of *FORM* for this. For *FORM* experts: all indices and vectors are declared D-dimensional, and a `trace4` (4-dimensional trace) command is issued (see [Ve91], specifically P. 189).

gives

$$-\frac{\texttt{EL}^2\ \texttt{ME}\ \texttt{B0[Pair[k[1], k[1]], ME2, 0]}}{4\ \texttt{Pi}^2} -$$

$$\frac{\texttt{EL}^2\ \texttt{B1[Pair[k[1], k[1]], ME2, 0]}\ \texttt{ga[k[1]]}}{8\ \texttt{Pi}^2} - \frac{\texttt{EL}^2\ \texttt{(-2 ME + ga[k[1]])}}{16\ \texttt{Pi}^2}$$

whereas with `$Dimension = 4` the result is just

$$-\frac{\texttt{EL}^2\ \texttt{ME}\ \texttt{B0[Pair[k[1], k[1]], ME2, 0]}}{4\ \texttt{Pi}^2} -$$

$$\frac{\texttt{EL}^2\ \texttt{B1[Pair[k[1], k[1]], ME2, 0]}\ \texttt{ga[k[1]]}}{8\ \texttt{Pi}^2}$$

Important: Since kinematical invariants, on-shell transformation rules and the like are set up the moment an amplitude is loaded, `OneLoop` must not be used on amplitudes of a certain process if in the meantime amplitudes belonging to a different process were loaded! The results are wrong if one tries to first load, for example, the electron and photon self-energy amplitudes and then simplify them with `OneLoop`. The correct sequence is to load the electron self-energies, simplify them with `OneLoop`, then load the photon self-energies, and simplify them with `OneLoop`.

1.3.3 Processing Entire Files

Even more comprehensive than `OneLoop`, the function `ProcessFile` can process entire files. It splits the results up into a bosonic and a fermionic part; this is commonly needed for the numerical evaluation e.g. if one wants to sum over fermion generations.

`ProcessFile[`*file*`,` *outname*`]`	process the amplitudes in *file* and write the results to files beginning with *outname*
`ProcessFile[`*amps*`,` *outname*`]`	process the amplitudes *amps* and write the results to files beginning with *outname*

For example,

```
ProcessFile["vertex.amp", "results/vertex"]
```

reads the *FeynArts* amplitudes in `vertex.m` and produces two files: `results/vertex.m` which contains the bosonic part and `results/vertexF.m` which contains the fermionic part.

`ProcessFile` is of course just a convenient way of invoking `OneLoop`. It therefore accepts all `OneLoop` options which it passes through to `OneLoop`.

option	*default value*	
`ToughStuff`	`False`	whether to further split up box contributions

Box diagrams (in particular fermionic ones) can easily produce output that is too large to be handled conveniently even after separating fermionic and bosonic parts. In such a case the option `ToughStuff -> True` should be used. Then, not only fermionic and bosonic diagrams are separated, but also different topologies and, for the fermionic diagrams, different fermion types (leptons, up-type quarks, and down-type quarks). The reason for this is that it turns out quite generally that only very few algebraic simplifications can be made across these so-allotted categories.

The files thus obtained typically have names like *outname*`FUT2.m` which indicates that this file contains the results of the fermionic (`F`) diagrams with only up-type quarks (`U`) from topology `T2`, where `T2` is the name that *FeynArts* assigns to the topology. The filenames are primarily constructed to distinguish the different files so one should not worry about them too much.

1.3.4 Handling Abbreviations

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

```
C0i[cc1, MW2, S, MW2, MZ2, MW2, MW2] *
  (P12*S*(-8*a2*MW2 + 4*a2*MW2*S2 - 28*a2*CW^2*MW2*S2 +
          16*a2*CW^2*S*S2 + 4*a2*C2*MW2*SW^2) +
   O47*S*(-32*a2*CW^2*MW2*S2 + 8*a2*CW^2*S2*T + 8*a2*CW^2*S2*U) -
   P13*S*(-64*a2*CW^2*MW2*S2 + 16*a2*CW^2*S2*T + 16*a2*CW^2*S2*U) +
   O3*(-2*a2*MW2*T + a2*MW2*S2*T - 7*a2*CW^2*MW2*S2*T +
       4*a2*CW^2*S*S2*T + a2*C2*MW2*SW^2*T + 2*a2*MW2*U -
       a2*MW2*S2*U + 7*a2*CW^2*MW2*S2*U - 4*a2*CW^2*S*S2*U -
       a2*C2*MW2*SW^2*U))
```

The first line stands for the tensor-coefficient function $C_1(M_W^2, s, M_W^2, M_Z^2, M_W^2, M_W^2)$ (see Sect. 2.3.4 for its definition) which is multiplied with a linear combination of abbreviations like `O47` or `P12` with certain coefficients.

These coefficients contain the Mandelstam variables `S`, `T`, and `U` and some short-hands for parameters of the Standard Model, e.g. `a2` = α^2, or `MW2` = M_W^2.[‖] They are collected in a way optimized for numerical evaluation: for example, *Mathematica* will automatically add together the terms in `-2*a2*MW2*T + a2*MW2*S2*T - 7*a2*CW^2*MW2*S2*T` once the numerical values for the various parameters are substituted (i.e. the result is of the form `(a number)*T`), so it is not worth simplifying this expression further.[**]

If desired, *FormCalc* can nevertheless perform even these last simplifications: *FormCalc* generally wraps two functions, `o1` and `o2`, around coefficients at different levels in the

[‖] It can be argued that *FormCalc* should not introduce model-dependent symbols. The advantage of doing so is however just too great to ignore as it can speed up calculations by as much as 15%. To look up, change, or add to these definitions, edit `FormCalc.m` and find the section beginning with the line

```
(* definitions specially for the Standard Model *)
```

In addition, because *FORM* works much more efficiently with polynomials than with rational expressions, four abbreviations for common negative powers of the sine and cosine of the weak mixing angle are introduced in `pave.frm`: `C4` = $\cos^{-4}\theta_W$, `C2` = $\cos^{-2}\theta_W$, `S4` = $\sin^{-4}\theta_W$, and `S2` = $\sin^{-2}\theta_W$.

[**] Global algebraic operations—operations that require to examine all terms of an expression simultaneously like factoring—can be very costly in time. For a good overall performance it is therefore important to weigh carefully whether a symbolic simplification reduces significantly enough the time to evaluate the expression numerically, i.e. whether it pays e.g. to factor the expression.

amplitude. The default setting `o1 = o2 = Identity` can be changed e.g. to `o1 = o2 = Simplify` to obtain the most compact analytical form.

object	*default value*	
`o1`	`Identity`	function wrapped around coefficients of abbreviations
`o2`	`Identity`	function wrapped around coefficients of one-loop functions
`$O1ME`	`1`	scale factor for abbreviations that contain momenta

Abbreviations are defined with the designated terms divided by `$O1ME` for every momentum they include. For instance, `Pair[e[1], k[2]]` would be divided by `$O1ME`, and `Eps[e[1], k[1], k[2], k[3]]` by `$O1ME`^{3}. The idea is to make the abbreviations dimensionless by cleverly choosing a scale factor, e.g. `$O1ME = Sqrt[S]` for a $2 \to 2$ process (the name `$O1ME` is actually derived from "matrix elements of order 1"). This can be of advantage when modifying the *FormCalc* output further. The powers of `$O1ME` introduced do not change the overall value of an amplitude, i.e. a candidate for abbreviationing is expanded like `Pair[e[1], k[1]] = $O1ME*`$\boxed{\texttt{Pair[e[1], k[1]]/\$O1ME}}$ before a new abbreviation is assigned to the expression in the box.

Important: do not (= DO NOT) change `$O1ME` between calls to `OneLoop`.

The abbreviations like `O47` or `P12` can drastically reduce the size of an amplitude. Spinor chains are abbreviated as `F`*nn*, but do not appear in the excerpt of code above since that is from a purely bosonic amplitude.

Consider `P13` from the example above, which is an abbreviation of about average length. It is composed of

```
P13 = O11 - O12 - O13 + O14 + O20
```

The `O`*nn* are in turn made up of

```
O11 = e23*k13*k43
O12 = e13*k23*k43
O13 = e13*k24*k43
O14 = e12*k31*k43
O20 = e23*k14*k43
```

and the e*nn* and k*nn* are defined as

```
e12 = Pair[e[1], e[2]]
e13 = Pair[e[1], e[3]]
k14 = Pair[e[1], k[4]]
...
```

To get an idea of how advantageous the introduction of abbreviations can be, it is useful to compare the `LeafCount` of the expressions in *Mathematica*. The leaf count gives a measure of the size of an expression, more precisely it counts the number of subexpressions or "leaves" on the expression tree. `P13` has a leaf count of 1 since it is just a plain symbol. In comparison, its fully expanded contents have a leaf count of 113.

`Abbreviations[]`	the list of abbreviations introduced so far
`OptimizeAbbreviations[`*a*`]`	optimize the abbreviations *a*

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

```
result //. Abbreviations[]
```

gives the full, unabbreviated expression. It is of course necessary to save the abbreviations before terminating a session, e.g. with

```
Abbreviations[] >> abbr
```

Abbreviations containing `Pair` or `Eps` are wrapped in `HoldForm` to prevent *Mathematica* from applying the transformation rules which turn them into abbreviations again. In a different *Mathematica* session the `HoldForm` can easily be removed with `ReleaseHold`.

`OptimizeAbbreviations[`*a*`]` optimizes the abbreviations *a*. It takes either the abbreviation rules as returned by `Abbreviations[]` or a file name. In the latter case the abbreviations are read from the file, optimized, and written back to the file, the original version of which is renamed to *file*`.orig` before.

The optimization is done in two steps. First, redundant parts are removed, e.g. the abbreviations

```
P637 -> O109 - O187
P504 -> O109 - O173 - O174 - O187
P566 -> O109 + O173 + O174 - O187
```

are replaced by

```
P637 -> O109 - O187
P504 -> P637 - O173 - O174
P566 -> P637 + O173 + O174
```

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

```
P637 -> O109 - O187
t1 -> O173 + O174
P504 -> P637 - t1
P566 -> P637 + t1
```

Optimizing the abbreviations can take a lot of time but can also speed up numerical computations considerably. Again, this is a question of weighing the overall savings.

1.4 A Demonstration Run

In the following a brief but characteristic excerpt of code for each step of a one-loop calculation as outlined in Sect. 1.1 is presented. The complete demonstration code is contained in the `demo` subdirectory of the *FormCalc* distribution.

As an example the elastic scattering of Z bosons in the electroweak Standard Model is chosen [DeDH97]. Two programs are supplied with the demo code to show how the numerical evaluation can be done:

- *NumPrep* is used to convert the *FormCalc* results into Fortran code,
- `num.F` is needed as a driver program for the Fortran code produced by *NumPrep*.

Although both were designed for $2 \to 2$ gauge-boson scattering processes, it should not be too difficult to adapt them at least to other $2 \to 2$ processes. (*NumPrep* is quite generic already; basically one has to supply the kinematics in `num.F`.)

Step 1: Generate the diagrams. In this code excerpt the amplitudes of the self-energy diagrams is generated (here *FeynArts* 2.2 is used).

```
<< FeynArts.m

tops = CreateTopologies[ 1, 2 -> 2,              (* 1 loop, 2 -> 2 *)
  ExcludeTopologies ->               (* create only self-energies *)
    {Tadpoles, WFCorrections, Triangles, AllBoxes} ]

inss = InsertFields[ tops,
  {V[2], V[2]} -> {V[2], V[2]},                        (* V[2] = Z *)
  Model -> "SM", InsertionLevel -> {Particles},
  Restrictions -> {NoGeneration2, NoGeneration3} ]

amps = CreateFeynAmp[inss];
ToFA1Conventions[amps] >> zzzz.self.amp
```

The procedure is typical for *FeynArts*: create the topologies, insert fields into them (`V[2]` is a Z boson in *FeynArts* lingo), and create the amplitudes, i.e. apply the Feynman rules. The function `ToFA1Conventions` converts the full *FeynArts* 2.2 format back into the simpler *FeynArts* 1 format which is needed by *FormCalc*.

Step 2: Simplify analytically. The so-created self-energy diagrams are now simplified with *FormCalc*. Calculating in D and 4 dimensions yields equivalent results in the case of $ZZ \to ZZ$.

```
<< FormCalc.m

counter = << zzzz.counter.amp;
ProcessFile[
  Join[ << zzzz.self.amp,
        Select[counter, DiagramType[#] == 2 &] ],
  "self" ]

Abbreviations[] >> abbr
```

The idea here is to calculate the *renormalized*[††] self-energy contribution, for which the appropriate counter-term diagrams have to be selected and joined with the virtual self-energy diagrams.

Step 3: Evaluate numerically. The *FormCalc* results need to be converted into a Fortran program. (The numerical evaluation could, in principle, be done in *Mathematica* directly, but this becomes very slow for large amplitudes.) The simplest way to do this in *Mathematica* is something like `FortranForm[result] >> file.f`. With the demo code a much more sophisticated program called *NumPrep* is supplied which goes well beyond simple translation into Fortran code. For example, it groups the one-loop integrals into angle-dependent and -independent integrals, so that the latter need to be calculated only once e.g. when integrating over the angle at a fixed energy.

NumPrep is too complex to describe here in detail. Instead, an example of the final Fortran code produced by *NumPrep* is given:

```
#include <defs.h>
        double complex function self()
        implicit logical (a-s,u-z)
        implicit double complex (t)
#include <vars.h>
        self = reso**2*(-5.747766329670332791488109665178O*ab61*O3 -
     -     3.69676282853998026863617405220037 2*ab62*O3 +
     -     ab59*(14.787051314159921074544696208801489 +
     -        0.000222288099036683807343741914688199 73*MH2)*O3 +
     -     ab58*(22.991065318681331165952438660711951 +
     -        0.000444576198073367614687483829376399 5*MH2)*O3 +
     -     0.000666864297110051422031225744064599 2*ab60*MH2*O3 +
     -     0.002000592891330154266093677232193797 6*ab15*MH2**2*
     -      O3 - ab14*O3*(-215176.58454926418740557299134203950 -
     -        0.000222288099036683807343741914688199 73*MH2**2 +
     -        7.393525657079960537272348104400745*S) - ...
```

Admittedly, this code looks ugly to the human eye, but is actually highly optimized for numerical evaluation: apart from the `O3` which is one of the abbrevia-

[††]In the implementation of the Standard Model in *FeynArts* the on-shell renormalization scheme of [De93] is used.

tions already introduced by *FormCalc*, the one-loop integrals have been replaced by variables (e.g. `ab`*nn* for A and B functions). Also, the possibly resonant Higgs propagator $1/(s - M_{\rm H}^2)$ has been replaced by the variable `reso` so that it can be treated more easily in Fortran. These are just examples of what one can do with the *FormCalc* output.

Of course, *NumPrep* produces also the code to calculate the abbreviations and a `Makefile` to conveniently compile the code. The Fortran code produced by *NumPrep* needs in addition a driver program which supplies it with the necessary parameters, kinematics, etc. This driver program is called `num.F` and is included in the demo.

2 *LoopTools*

LoopTools is a package for evaluation of scalar and tensor one-loop integrals based on the *FF* package by G.J. van Oldenborgh [vOV90]. It provides the actual numerical implementations of the functions appearing in *FormCalc* output. These are the scalar one-loop functions of *FF* and the 2-, 3-, and 4-point tensor-coefficient functions in the conventions of [De93]. *LoopTools* offers three interfaces, Fortran, C++, and *Mathematica*, so most programming tastes should be served.

2.1 Making the Libraries

To compile the package a Fortran-77 compiler, the GNU C and C++ compilers (`gcc` and `g++`), and GNU's `make` utility (sometimes called `gmake`) are required. Due to differences in conventions between versions, the package is designed 'as is' to compile with `g++` version 2.7.2 or higher. Before attempting to compile the *LoopTools* libraries yourself, check out the *LoopTools* web page (`http://www-itp.physik.uni-karlsruhe.de/looptools`) to see if there are any pre-compiled binaries available for your platform.

LoopTools comes in a compressed tar archive. It includes the *FF* package in a version certified to compile properly with the *LoopTools* makefile. A newer version might be available from `ftp://ftp.nikhef.nl/pub/ff/FF.html`, but beware that there may be changes which affect the compilation. *LoopTools* applies some patches to *FF* and adds two functions, the D_0 function for all four masses zero (`ffxd0m0.f`) and the derivative of the B_1 function (`ffxdb1.f`).

Choose a directory for *LoopTools*—`LT12`, say—and execute the following commands to unpack and compile the package.

```
mkdir LT12
cd LT12
gunzip -c ../LoopTools12.tar.gz | tar xvf -
./configure
gmake
gmake clean
```

The `configure` script will try to find out the necessary system information for the compilation. `gmake` then makes the following objects:

`libff.a`	the (modified) *FF* library for Fortran
`bca`	the *MathLink* executable for *Mathematica*
`libhep.a`	the tensor function library for C++
`libffx.a`	the quadruple precision *FF* library (as yet only on DEC Alpha platforms)

The libraries have static linkage by default. Invoking `gmake SHARED=`*shlibdir* will attempt to build shared libraries where *shlibdir* is the directory in which the shared objects will finally be installed, e.g. `/usr/local/lib`. Unfortunately, the Fortran compiler flags for shared compilation are not much standardized (usually `-shared`, but e.g. `-n` for the `f77` of HP-UX). This is something `configure` cannot determine so you might have to change the `GNUmakefile` yourself (look for the comments on `SHARED`).

The resulting directory structure should be

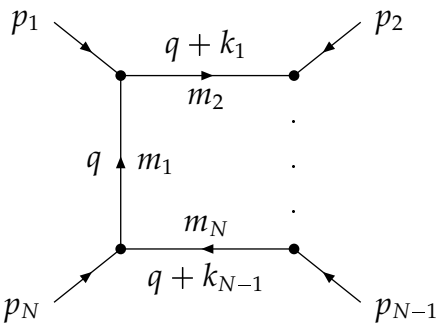
`LT12/`	the *LoopTools* directory
`LT12/$HOSTTYPE`	directory for the libraries and executables
`LT12/include`	directory for the include files
`LT12/ff`	a temporary directory during compilation (removed by `gmake clean`)

`HOSTTYPE` is an environment variable that is set by most shells to a string identifying the system, e.g. `alpha`, `hp9000s700`, or `i386-linux`. Its purpose as a directory name is to separate the binaries for different platforms.

In contrast to the original *FF* library, the *LoopTools* libraries and executables depend on no additional files (error message catalogues etc.), so they may be copied to some "public" place instead of `LT12/$HOSTTYPE`, e.g. the `bca` executable to `/usr/local/bin` and the libraries to `/usr/local/lib`. Superuser privileges are usually required for this.

2.2 One-Loop Integrals

Consider the following general one-loop diagram.



The integral contained in this diagram is

$$T^N_{\mu_1\ldots\mu_P} = \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2}\int \mathrm{d}^D q\, \frac{q_{\mu_1}\cdots q_{\mu_P}}{\left[q^2-m_0^2\right]\left[(q+k_1)^2-m_1^2\right]\cdots\left[(q+k_{N-1})^2-m_N^2\right]} \tag{2.1}$$

where the momenta k_i that appear in the denominators are related to the external momenta p_i as

$$\begin{aligned} &p_1 = k_1\,, \qquad p_2 = k_2 - k_1\,, \qquad \ldots \qquad p_N = k_N - k_{N-1}\,, \\ &k_1 = p_1\,, \qquad k_2 = p_1 + p_2\,, \qquad \ldots \qquad k_N = \sum_{i=1}^{N} p_i\,. \end{aligned} \tag{2.2}$$

The representation given in Eq. (2.1) is correct for dimensional regularization or dimensional reduction. (In the latter case the integrals are kept D-dimensional although the rest of the algebra is performed in 4 dimensions.) μ plays the rôle of a renormalization scale that keeps track of the correct dimension of the integral in D space–time dimensions. In constrained differential renormalization the mass scale enters in a conceptually different way; however, the dependence of the one-loop integrals on μ is the same as for dimensional regularization (for details see [HaP98]).

The denominators arise from the propagators running in the loop. P, the number of q's in the numerator, determines the Lorentz tensor structure of the whole integral, i.e. $P = 0$ denotes a scalar integral, $P = 1$ a vector integral, etc. From the definition it is obvious that the integrals are symmetric under permutation of the Lorentz indices. The q's in the numerator arise typically from fermion propagators or from vertices that correspond to terms with derivatives in the Lagrangian.

The nomenclature is A for T^1, B for T^2, etc. The scalar integrals are denoted by a subscripted zero: A_0, B_0, etc.

2.2.1 Tensor Coefficients

The integrals with a tensor structure can be reduced to linear combinations of Lorentz-covariant tensors constructed from the metric tensor $g_{\mu\nu}$ and a linearly independent set of the momenta [PaV79]. The choice of this basis is not unique.

LoopTools provides not the tensor integrals themselves, but the coefficients of these Lorentz-covariant tensors. It works in a basis formed from $g_{\mu\nu}$ and the momenta k_i, which are the sums of the external momenta p_i (see Eq. (2.2)) [De93]. In this basis the tensor-coefficient functions are totally symmetric in their indices. For the integrals up to the four-point function the decomposition reads explicitly

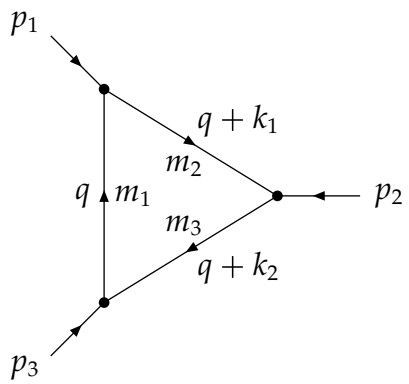
$$
\begin{aligned}
B_\mu &= k_{1\mu} B_1 \,, \\
B_{\mu\nu} &= g_{\mu\nu} B_{00} + k_{1\mu} k_{1\nu} B_{11} \,, \\
C_\mu &= k_{1\mu} C_1 + k_{2\mu} C_2 = \sum_{i=1}^{2} k_{i\mu} C_i \,, \\
C_{\mu\nu} &= g_{\mu\nu} C_{00} + \sum_{i,j=1}^{2} k_{i\mu} k_{j\nu} C_{ij} \,, \\
C_{\mu\nu\rho} &= \sum_{i=1}^{2} \big(g_{\mu\nu} k_{i\rho} + g_{\nu\rho} k_{i\mu} + g_{\mu\rho} k_{i\nu} \big) C_{00i} + \sum_{i,j,\ell=1}^{2} k_{i\mu} k_{j\nu} k_{\ell\rho} C_{ij\ell} \,, \\
D_\mu &= \sum_{i=1}^{3} k_{i\mu} D_i \,, \\
D_{\mu\nu} &= g_{\mu\nu} D_{00} + \sum_{i,j=1}^{3} k_{i\mu} k_{j\nu} D_{ij} \,, \\
D_{\mu\nu\rho} &= \sum_{i=1}^{2} \big(g_{\mu\nu} k_{i\rho} + g_{\nu\rho} k_{i\mu} + g_{\mu\rho} k_{i\nu} \big) D_{00i} + \sum_{i,j,\ell=1}^{2} k_{i\mu} k_{j\nu} k_{\ell\rho} D_{ij\ell} \,, \\
D_{\mu\nu\rho\sigma} &= (g_{\mu\nu} g_{\rho\sigma} + g_{\mu\rho} g_{\nu\sigma} + g_{\mu\sigma} g_{\nu\rho}) D_{0000} \\
&\quad + \sum_{i,j=1}^{3} \big(g_{\mu\nu} k_{i\rho} k_{j\sigma} + g_{\nu\rho} k_{i\mu} k_{j\sigma} + g_{\mu\rho} k_{i\nu} k_{j\sigma} \\
&\qquad + g_{\mu\sigma} k_{i\nu} k_{j\rho} + g_{\nu\sigma} k_{i\mu} k_{j\rho} + g_{\rho\sigma} k_{i\mu} k_{j\nu} \big) D_{00ij} \\
&\quad + \sum_{i,j,\ell,m=1}^{3} k_{i\mu} k_{j\nu} k_{\ell\rho} k_{m\sigma} D_{ij\ell m} \,.
\end{aligned}
$$

Of all scalar and tensor-coefficient functions only A_0, B_0, B_1, B_{00}, B_{11}, C_{00}, C_{001}, C_{002}, and D_{0000} are actually UV divergent.

2.2.2 Conventions for the Momenta

A large source of mistakes is the way of specifying the momenta in the one-loop integrals. The prime error in this respect is the confusion of the external momenta p_i with the momenta k_i appearing in the denominators, which are the sums of the p_i (see Eq. (2.2)).

Consider for example the following diagram:



The three-point function corresponding to this diagram can be written either in terms of the external momenta as

$$C\big(p_1^2, p_2^2, (p_1 + p_2)^2, m_1^2, m_2^2, m_3^2\big)$$

or in terms of the momenta k_i as

$$C\big(k_1^2, (k_1 - k_2)^2, k_2^2, m_1^2, m_2^2, m_3^2\big)\ .$$

In both cases the *same* function is called with the *same* arguments since of course $k_1 = p_1$ and $k_2 = p_1 + p_2$. (The arguments are given in the conventions of *LoopTools*.)

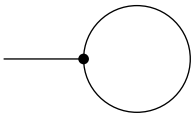
It is however important to realize that *LoopTools* functions like C_1 and C_{112} are the coefficients respectively of $k_{1\mu}$ and $k_{1\mu}k_{1\nu}k_{2\rho}$, not of $p_{1\mu}$ and $p_{1\mu}p_{1\nu}p_{2\rho}$.

2.3 Functions provided by *LoopTools*

2.3.1 One-point function

Function call	Description
`A0(ms)`	one-point function

The real argument `ms` (`double precision` in Fortran) is the mass squared:

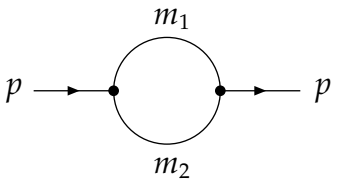


$$= \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2} \int \frac{\mathrm{d}^D q}{q^2 - m^2} \qquad \boxed{\texttt{ms} = m^2}$$

2.3.2 Two-point functions

Function call	Description
`B0(ps, m1s, m2s)`	scalar two-point function
`B1(ps, m1s, m2s)`	coefficient of p_μ
`B00(ps, m1s, m2s)`	coefficient of $g_{\mu\nu}$
`B11(ps, m1s, m2s)`	coefficient of $p_\mu p_\nu$

All arguments are real (`double precision` in Fortran) and are related to the momenta and masses as follows:



$$= \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2} \int \frac{(\text{numerator})\ \mathrm{d}^D q}{\left[q^2 - m_1^2\right]\left[(q+p)^2 - m_2^2\right]}$$

$$\boxed{\begin{aligned} \texttt{ps} &= p^2 \\ \texttt{m1s} &= m_1^2 \\ \texttt{m2s} &= m_2^2 \end{aligned}}$$

2.3.3 Derivatives of Two-point functions

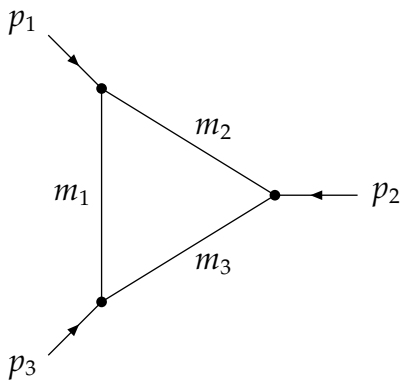
Function call	Description
`DB0(ps, m1s, m2s)`	derivative of `B0`
`DB1(ps, m1s, m2s)`	derivative of `B1`
`DB00(ps, m1s, m2s)`	derivative of `B00`
`DB11(ps, m1s, m2s)`	derivative of `B11`

All derivatives are with respect to the momentum squared. The arguments are the same as for the two-point functions.

2.3.4 Three-point functions

Function call	Description
`C0(p1s, p2s, p1p2s, m1s, m2s, m3s)`	scalar three-point function
`C0i(id, p1s, p2s, p1p2s, m1s, m2s, m3s)`	three-point tensor coefficients

Except for the `id` all arguments are real (`double precision` in Fortran) and are related to the momenta and masses as follows:



$$\begin{aligned}
\texttt{p1s} &= p_1^2 \\
\texttt{p2s} &= p_2^2 \\
\texttt{p1p2s} &= (p_1+p_2)^2 \\
\texttt{m1s} &= m_1^2 \\
\texttt{m2s} &= m_2^2 \\
\texttt{m3s} &= m_3^2
\end{aligned}$$

$$= \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2} \int \frac{(\text{numerator})\ \mathrm{d}^D q}{\left[q^2-m_1^2\right]\left[(q+p_1)^2-m_2^2\right]\left[(q+p_1+p_2)^2-m_3^2\right]}$$

`C0i` is a generic function for all three-point tensor coefficients. A specific coefficient is selected with the `id` argument:

$$\begin{aligned}
\texttt{C0i(cc0, ...)} &= C_0(\ldots) \\
\texttt{C0i(cc00, ...)} &= C_{00}(\ldots) \\
\texttt{C0i(cc112, ...)} &= C_{112}(\ldots) \qquad \text{etc.}
\end{aligned}$$

Since the indices are symmetric, the identifiers are assumed to be ordered, i.e. there is only `cc122` but not `cc212`.

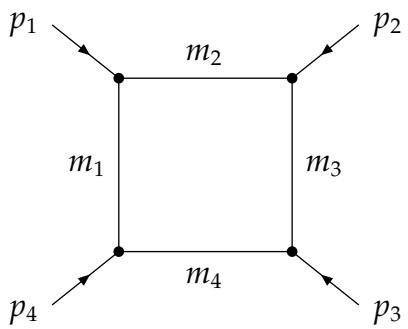
Internally, what happens when `C0i` is called is that actually *all* coefficients (with the respective arguments `p1s`, `p2s`, ...) are calculated. This is because there are a lot of intermediate results which would have to be recalculated every time the function is called for a different coefficient. These coefficients are then of course stored so that repeated

calls to `C0i` or `D0i` with the same set of arguments will simply retrieve the value from memory. So in a very real sense the identifiers `cc0`, `cc001`, etc. can be thought of as array indices (in fact, they are just integer constants to the compiler). In an unoptimized program, the savings incurred by this mechanism can be sizeable: typically 90% of integrals requested can be retrieved from cache.

2.3.5 Four-point functions

Function call	Description
`D0(p1s, p2s, p3s, p4s, p1p2s, p2p3s, m1s, m2s, m3s, m4s)`	scalar four-point function
`D0i(id, p1s, p2s, p3s, p4s, p1p2s, p2p3s, m1s, m2s, m3s, m4s)`	four-point tensor coefficients

Except for the `id` all arguments are real (`double precision` in Fortran) and are related to the momenta and masses as follows:



$\texttt{p1s} = p_1^2$	$\texttt{m1s} = m_1^2$
$\texttt{p2s} = p_2^2$	$\texttt{m2s} = m_2^2$
$\texttt{p3s} = p_3^2$	$\texttt{m3s} = m_3^2$
$\texttt{p4s} = p_4^2$	$\texttt{m3s} = m_4^2$
$\texttt{p1p2s} = (p_1 + p_2)^2$	
$\texttt{p2p3s} = (p_2 + p_3)^2$	

$$= \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2} \int \frac{(\text{numerator})\, \mathrm{d}^D q}{\left[q^2 - m_1^2\right]\left[(q+p_1)^2 - m_2^2\right]\left[(q+p_1+p_2)^2 - m_3^2\right]\left[(q+p_1+p_2+p_3)^2 - m_4^2\right]}$$

`D0i` is a generic function for all four-point tensor coefficients. A specific coefficient is selected with the `id` argument:

$$\begin{aligned} \texttt{D0i(dd0, ...)} &= D_0(\ldots) \\ \texttt{D0i(dd003, ...)} &= D_{003}(\ldots) \\ \texttt{D0i(dd1223, ...)} &= D_{1223}(\ldots) \qquad \text{etc.} \end{aligned}$$

Again, since the indices are symmetric, the identifiers are assumed to be ordered, i.e. there is only `dd1223` but not `dd3212`. Similar to `C0i` the results of `D0i` are also cached in memory (see end of Sect. 2.3.4).

2.3.6 On Warning Messages

The warnings issued by *FF* concerning the loss of accuracy are somewhat overzealous, and particularly for a large number of consecutive calls to *FF* (e.g. when computing a cross-section over a sizeable region of phase space) can add up to ridiculous numbers, e.g. "lost a factor 10^5." Unless a very detailed checking of these warnings is performed, they are pretty useless and tend to numb the user to a degree where severe errors are easily overlooked. For this reason, the *FF* warning system has been disabled in *LoopTools*. That is not to say that one-loop computations have no numerical problems, only that the warning system is useful for detecting them only under certain circumstances. Whatever problems are still reported by *FF* should be considered serious!

"Unsmoothness" of a curve (unexpected bumps or peaks) is generally a tell-tale signal of numerical problems. Problematic regions are typically thresholds and high energies.

2.4 Using *LoopTools* with Fortran

Some technical details concerning compilation:

- When compiling a program that uses *LoopTools*, add

  ```
  -I$HOME/LT12/include -L$HOME/LT12/$HOSTTYPE -lff
  ```

  to the `f77` command line.

- The `.F` programs described in the following use the C preprocessor. Most Fortran compilers recognize this by the capital `.F`; if yours is not that smart, use the script `~/LT12/utils/F77` instead of the normal `f77`.

- If you have to use `f2c` and `cc` (at the moment still a better choice than `g77`), the script `~/LT12/utils/f77c` is useful to compile Fortran programs with one command.

- To have access to the common blocks of the *FF* routines directly, include `ff.h`.

To use the *LoopTools* functions in a Fortran program, the two files `tools.F` and `tools.h` must be included. `tools.F` contains actual Fortran code and should be included only once per program. `tools.h` contains the declarations and common blocks and must be included in every function or subroutine in which the *LoopTools* functions are used.

If the program consists of several modules of code, include `tools.F` only in one module, include `defs.h` in the others instead. For example,

program	*include once*	*include in every subroutine/function*
`main.F`	`tools.F`	`tools.h`
`sub1.F`	`defs.h`	`tools.h`
`sub2.F`	`defs.h`	`tools.h`

Before using any *LoopTools* function, the subroutine `bcaini` must be called. At the end of the calculation `bcaexi` may be called to obtain a summary of errors.

A very elementary program would for instance be

```
#include "tools.F"

        program simple_program
#include "tools.h"

        call bcaini
        print *, B0(1000D0, 50D0, 80D0)
        call bcaexi
        end
```

Note that, as for all preprocessor commands, the `#` must stand at the beginning of the line. It is important to include the files via the preprocessor commands instead of the `include` directive many Fortran compilers offer. This is because the options described in the following are all chosen with preprocessor variables which would otherwise not take effect.

`tools.F` uses UV- and IR-regularization parameters which may be varied to test UV- and IR-finiteness. The square of the renormalization scale μ used in the divergent functions may be specified (*before* including `tools.F`) by

```
#define MUDIM ...
```

and the square of the regulator mass λ used in IR-divergent functions by

```
#define LAMBDA ...
```

If not defined, `MUDIM` and `LAMBDA` are both set to 1.0. Note that both `MUDIM` and `LAMBDA` enter logarithmically. This means that to decisively check whether an expression is really independent of them, they must be varied on a large scale, e.g. from 1 to 10^{20}.

The quantity $\Delta = -2/(D-4) - \gamma_E + \log 4\pi$ which parametrizes the infinity in dimensional regularization is set to 0, the $\overline{\text{MS}}$ value. Putting $\Delta = -2$ would reproduce the one-loop functions of CDR as published in [dACTP98]. Δ is actually a redundant parameter since the renormalization scale can be adjusted to have the same effect: $\mu^2_{\text{new}} = e^{\Delta}\mu^2_{\text{old}}$. In an on-shell renormalization scheme the dependence on Δ must drop out altogether.

The reason why `tools.F` must be included instead of being part of the library is that standard Fortran 77 does not support dynamical memory allocation. Therefore, the arrays in which `C0i` and `D0i` store their results (see Sect. 2.3.4) are set up in static memory and you must first tell `tools.F` roughly how much storage you will need. For example,

```
#define CSTORE 400
#define DSTORE 200
#include "tools.F"
```

will set up arrays large enough to contain 400 sets of results from `C0i` and 200 from `D0i`. Note that a call to `D0i` invokes four calls to `C0i`. If either bound is transgressed, `tools.F` will issue a runtime warning. The results will be reliable in any case, only the advantage in speed is lost. If not defined, `CSTORE` and `DSTORE` are set to 170 and 30, respectively.

Even though these internal arrays can speed up calculations by a large factor, flushing them at the right time can be a very clever way to save on memory. For instance, you may have a loop like

```
do Energy = 100D0, 1000D0, 50D0
  (... calculations involving C0i and D0i ...)
enddo
```

Usually the parameters of `C0i` and `D0i` depend on the energy, so the odds are that none of the previously calculated results can be reused at a different energy. To discard the old results and make room for new ones you can reset the array pointers: include `tools.h` in the declarations section (for the common blocks) and then say

```
Cptr = 1
Dptr = 1
```

An even more advanced version would be this: Suppose you have a total cross-section to calculate where some integrals depend on the angular variables and others do not. Then you might use a construct like

```
        integer Cptr2, Dptr2

        do Energy = 100D0, 1000D0, 50D0
          (... calculate angle-independent integrals ...)
          Cptr2 = Cptr
          Dptr2 = Dptr
          do theta = cutoff, pi - cutoff, pi/180D0
            (... calculate angle-dependent integrals ...)
            Cptr = Cptr2
            Dptr = Dptr2
          enddo
          Cptr = 1
          Dptr = 1
        enddo
```

Finally, to give a real-life example, here is the calculation of the bosonic part of the Higgs self-energy in the electroweak Standard Model.

```
#include "tools.F"

        program HiggsSE
#include "tools.h"
        double precision s
        double complex SigmaH
        external SigmaH

        call bcaini
        do s = 100D0, 1000D0, 50D0
          print *, s, "  ", SigmaH(s)
        enddo
        call bcaexi
        end
```

```
        double complex function SigmaH(k2)
        double precision k2
#include "tools.h"
        double precision MH2, MZ2, MW2, alpha, pi, SW2
        parameter (MH2 = 100D0**2,
     +    MZ2 = 91.188D0**2,
     +    MW2 = 80.4D0**2,
     +    alpha = 1D0/137.0359895D0,
     +    pi = 3.14159265358979D0,
     +    SW2 = 1D0 - MW2/MZ2)

        SigmaH = alpha/32D0/pi/SW2/MW2*
     +    ( 3D0*MH2*A0(MH2) + 9D0*MH2**2*B0(k2, MH2, MH2)
     +    + 2D0*(MH2**2 - 4D0*MW2*(k2 - 3D0*MW2))*B0(k2, MW2, MW2)
     +    + 2D0*(6D0*MW2 + MH2)*A0(MW2) - 24D0*MW2**2
     +    + (MH2**2 - 4D0*MZ2*(k2 - 3D0*MZ2))*B0(k2, MZ2, MZ2)
     +    + (6D0*MZ2 + MH2)*A0(MZ2) - 12D0*MZ2**2 )
        end
```

2.5 Using *LoopTools* with C++

Some technical details:

- Compile programs that use *LoopTools* with the following command line:

  ```
  ~/LT12/utils/ccf -I$HOME/LT12/include (source files) \
     -L$HOME/LT12/$HOSTTYPE -lhep -lff
  ```

  `ccf` is a script to compile C and C++ programs and link them with Fortran libraries, in this case `libff.a`.

- In bare C there is no intrinsic type `complex`, and using things like `structs` is rather annoying. Therefore, I have made no attempt to implement the tensor package in bare C. Unfortunately, even in C++ the type `complex` is anything but standardized. As it stands, the package will compile with `g++` 2.7.2 and since no templates are

used, there is hope it will do so also under higher versions of `g++`. Using any non-GNU C++ compiler is at your own risk. If you really must work with a different compiler, the most likely cause of problems is the different naming convention for complex variables. If so, the following compiler switch will help:

```
-Ddouble_complex=mycomplexname
```

where `mycomplexname` is your compiler's name for the complex data type, usually something like `complex`, `Complex` etc.

- The three- and four-point tensor functions use the quadruple-precision package `math++` by Keith Briggs in intermediate calculations. This package is available at `http://epidem13.plantsci.cam.ac.uk/~kbriggs/math++1.9`.

To use the *LoopTools* functions in a C++ program, the file `ctools.h` must be included. Similar to the Fortran case, before making the first call to any *LoopTools* function, `bcaini()` must be called and at the end `bcaexi()` may be called to get a summary of errors.

In C++, an elementary program would be

```
#include <fstream.h>                  // for cout
#include "ctools.h"

main()
{
  bcaini();
  cout << B0(1000., 50., 80.) << endl;
  bcaexi();
}
```

To change the square of the renormalization scale μ taken in the divergent functions use

```
set_mudim(new_mudimsquared)
```

This can be useful when checking whether an expression is UV-finite. The square of the regulator mass λ used in IR-divergent functions can be set with

```
set_lambda(new_lambdasquared)
```

Both regularization parameters are initially set to 1.0 by `bcaini`. As already mentioned in Sect. 2.4, μ and λ must be varied by a big factor to decisively check independence because they enter logarithmically. Δ is fixed at 0 as in the Fortran case.

For the caching of the `C0i` and `D0i` results no array bounds need to be specified as in Fortran. Depending on the machine there is usually enough memory to handle quite a lot of integrals. Nevertheless, I'm afraid there is no check for overflow; in the event the program will probably crash—it never occurred to me even in very large calculations.

Flushing the internal heap can be a good idea to save on memory also in C++. This is done by calling

```
resetheap()
```

In the following the same example as for the Fortran case is given: the bosonic part of the Higgs self-energy in the electroweak Standard Model. At first sight, the C++ code looks more appealing, but remember that `double_complex` is not an intrinsic data type in C++ and that certainly compilation time and usually also the performance of the resulting program will be worse than in Fortran.

```
#include <std/complex.h>
#include <fstream.h>
#include "ctools.h"

double_complex SigmaH(double);

main()
{
  double s;

  bcaini();
  for(s = 100.; s <= 1000.; s += 50.)
    cout << s << "\t" << SigmaH(s) << endl;
  bcaexi();
}

#define MH2 (100.*100.)
#define MZ2 (91.188*91.188)
```

```
#define MW2 (80.4*80.4)
#define alpha (1./137.0359895)
#define pi 3.14159265358979
#define SW2 (1. - MW2/MZ2)

double_complex SigmaH(double k2)
{
  return alpha/32./pi/SW2/MW2*
    ( 3.*MH2*A0(MH2) + 9.*MH2*MH2*B0(k2, MH2, MH2)
      + 2.*(MH2*MH2 - 4.*MW2*(k2 - 3.*MW2))*B0(k2, MW2, MW2)
      + 2.*(6.*MW2 + MH2)*A0(MW2) - 24.*MW2*MW2
      + (MH2*MH2 - 4.*MZ2*(k2 - 3.*MZ2))*B0(k2, MZ2, MZ2)
      + (6.*MZ2 + MH2)*A0(MZ2) - 12.*MZ2*MZ2 );
}
```

2.6 Using *LoopTools* with *Mathematica*

Modify your path to include `~/LT12/$HOSTTYPE`, e.g. in `tcsh` use

```
set path=($path $HOME/LT12/$HOSTTYPE)
```

It is probably a good idea to include this statement e.g. in `.cshrc`.

The *Mathematica* interface is probably the simplest to use:

```
In[1]:= Install["bca"]

Out[1]= LinkObject[bca, 1, 1]

In[2]:= B0[1000, 50, 80]

Out[2]= -4.40593 + 2.70414 I
```

One-loop functions containing non-numeric arguments (e.g. `B0[1000, MW2, MW2]`) remain unevaluated. If it becomes necessary to switch off the evaluation of the *LoopTools* functions, `bca` can be uninstalled:

```
In[10^37]:= Uninstall[%1]
```

The square of the renormalization scale μ can be changed with

```
Mudim[new_mudimsquared]
```

and the square of the IR regulator mass λ with

```
Lambda[new_lambdasquared]
```

Both functions return the previous value of the respective parameter.

The value of Δ is 0 as in the Fortran case (cf. Sect. 2.4).

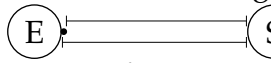
The internal caches for `C0i` and `D0i` (see Sects. 2.3.4 and 2.4) can be emptied with

```
FlushCache[]
```

Invoking `FlushCache[]` is however only necessary when large numbers of three- and four-point integrals are requested to avoid warning messages.

2.7 Remarks on Quadruple Precision

LoopTools includes an extended precision library. At the moment, this library will only compile on DEC Alpha platforms under Digital Unix 4.0 (the f90 compiler of older Digital Unix versions has some problems). There are no fundamental difficulties involved in porting to other platforms, only I have not had time to figure out various (non-standard) compiler switches for other vendors' f90.

Extended precision (`REAL*16` and `COMPLEX*32`) is considered by some people to be brute force, but becomes necessary e.g. in calculations with large gauge cancellations. The problem is the order of magnitude involved which is roughly comparable to measuring the thickness of a pencil by taking the distance earth–sun and subtracting the distance between the sun and the pencil on the earth's surface: 

. This can be particularly painful e.g. for high energies in the presence of corrections which are unitary as a whole but whose individual contributions grow as s or even s^2.

There are in principle two solutions:

a) use higher precision, or, if that is not available,

b) calculate a quantity once and subtract that value in the second pass thus eliminating large cancellations. E.g. if one stores the integer part of a Feynman diagram in, say, an `INTEGER*8` (which has a range of approximately $\pm 10^{19}$) then the large cancellations can be computed exactly in fixed-point arithmetic, and for the fractional parts double precision suffices.

The point is that b) requires extra programming and runs twice as slow since the calculation is done once to get an estimate of the true value, and then again to refine that estimate.

Of course an extended precision calculation also takes more time than a double precision calculation (but normally not twice as much), particularly so because extended precision is almost always emulated by a software library. What is a bit surprising is that a lot compilers support `REAL*16` but only very few (as far as I know some VAX and IBM compilers) offer `COMPLEX*32`, and doing mixed processing (real in quadruple and complex in double precision) yields almost no net improvement in accuracy.

The library `libffx.a` is therefore implemented in Fortran 90 using `REAL*16` and a new class `complex32`. To make Fortran-77 programs work with this library, the program `f77290` is supplied which converts fixed-style f77 input to free-style f90 input and takes care of the proper declarations for `complex32`.

Here is the example of the bosonic part of the Higgs self-energy in the electroweak Standard Model in Fortran-90 style:

```
#include "tools.f90"

program HiggsSE
#include "tools.h90"
real*16 :: s

call bcaini
do s = 100q0, 1000q0, 50q0
  print *, s, "  ", SigmaH(s)
enddo
call bcaexi

contains
```

```
function SigmaH(k2)
use qcomplex
implicit none
real*16, intent(in) :: k2
type(complex32) :: SigmaH
#include "tools.h90"
real*16, parameter :: MH2 = 100q0**2
real*16, parameter :: MZ2 = 91.188q0**2
real*16, parameter :: MW2 = 80.4q0**2
real*16, parameter :: alpha = 1q0/137.0359895q0
real*16, parameter :: SW2 = 1q0 - MW2/MZ2
real*16, parameter :: pi = 3.14159265358979q0

SigmaH = alpha/32q0/pi/SW2/MW2* &
  ( 3q0*MH2*A0(MH2) + 9q0*MH2**2*B0(k2, MH2, MH2) &
  + 2q0*(MH2**2 - 4q0*MW2*(k2 - 3q0*MW2))*B0(k2, MW2, MW2) &
  + 2q0*(6q0*MW2 + MH2)*A0(MW2) - 24q0*MW2**2 &
  + (MH2**2 - 4q0*MZ2*(k2 - 3q0*MZ2))*B0(k2, MZ2, MZ2) &
  + (6q0*MZ2 + MH2)*A0(MZ2) - 12q0*MZ2**2 )
end function SigmaH

end program HiggsSE
```

Actually, this program was written by hand in Fortran 90, and not converted by `f77290`, in order to show the full spectre of differences and enhancements of Fortran 90. In reality, f90 allows a more relaxed syntax, e.g. variables may be declared in the old way (`real*16 x` instead of `real*16 :: x`), modifiers like `intent(in)` are not mandatory, etc. Furthermore, numerical constants may be given in single or double precision notation (`1.234` or `1.234E0` or `1.234D0`) if the appropriate compiler switches are used.

The two most important differences are that at the beginning of each function `use qcomplex` is needed to enable the quadruple-precision complex type, and the actual complex numbers are declared by `type(complex32) ::` *var*.

To sum up: If you have a DEC Alpha and a Fortran-77 program that uses *LoopTools*, run the program including all header files through `f77290` and then compile it with `f90` using this sequence:

```
~/LT12/utils/f77290 myfile.f
f90 -real_size 128 -double_size 128 -cpp -I$HOME/LT12/include \
   myfile.f90 -L$HOME/LT12/$HOSTTYPE -lffx
```

One as yet unsolved problem are f77-style statement functions, which are not permitted with the `complex32` data type.

A The original *FF* Manual

A.1 Introduction

The evaluation of scalar loop integrals is one of the time consuming parts of radiative correction computations in high energy physics. Of course the general solution has long been known [tHV79], but the use of these formulae is not straightforward. If one encodes the algorithms directly in a numerical language one finds that for most physical configurations the answer is extremely unreliable due to numerical cancellations. It is not at all difficult to find examples where more than 80 digits accuracy are lost.

There are two ways in which these problems have been solved. M. Veltman has programmed these algorithms using a very large precision (up to 120 digits) for the intermediate results in the program FormF, which enabled him to do some very complicated calculations [PaV79]. However, these routines are written in assembler language and thus only available on certain computers. Also, the use of multiple precision makes them fairly slow — and even so there are many (soft t-channel) configurations for which the answer is incorrect, or correct only for one permutation of the input parameters. The other solution is to evaluate by hand all special cases needed and make sure that these are numerically stable, in this way building a library of physically interesting cases. This costs much time and has to be extended for every new calculation, as often the limits taken are no longer valid.

We present here a set of Fortran routines that evaluate the one-loop scalar integrals using a standard precision. The algorithms used have been published before [vOV90]. This paper describes version 1.0 which contains the following units:

- the scalar one, two, three, four and five-point functions, defined by

$$X_0 = \frac{1}{i\pi^2} \int \frac{d^n Q}{(Q^2 - m_1^2)((Q+P)^2 - m_2^2) \cdots} \tag{A.1}$$

- the vector three and four-point functions,
- some determinants.

Planned additions are:

- The other Form factors à la FormF.

- The six-point function.

Note however, that the reduction of these can be done analytically.

The aim of the routines is to provide a reliable answer for any conceivable (physical) combination of input parameters. This has not been fully met in the case of the four-point function, but an impressive list of cases does indeed work. Problems normally occur when many parameters are (almost) equal, i.e. when an analytical calculation is most feasible.

The layout of this paper is as follows. First we give a brief description of the design of the package and some details that may be of of relevance to the user, like timings. Next we give a complete user's guide. The problems which might be encountered when installing FF on a computer system are discussed in section A.3. The initialisation of the routines, which has to be done by the user in the program which uses the FF routines, is outlined in section A.4. The next section is about the use of the error reporting facilities, which also need some assistance from the user. A list of the available routines for the scalar n-point functions (section A.6) and determinants (section A.8) is given, listing parameters, loss of precision and comments.

A.2 Brief description of the scalar loop routines

This section will give an overview of the structure of the scalar loop routines which implement the algorithms of [vOV90]. The purpose of this is to provide a map for the adventurous person who wants to understand what is going on. Some details of the algorithms chosen are also given.

A.2.1 Overview

The language chosen is Fortran, mainly because so much of the calculations are done with complex variables. There are currently about 26000 lines of code. Some of it is repetitious, as many routines exist in a real and complex version which hardly differ. Global names (subprograms, common blocks) almost all start with the letters `FF`, for FormFactor (the only exceptions are the functions `dfflo1`, `zfflo1`, `zfflog` and `zxfflg`). For this reason I refer to the set as the FF package. The third letter of the name often indicates whether a routine is complex (`z` or `c`) or real. The real four-point function is

thus calculated with the routine `ffxd0`, the complex dilogarithm in `ffzli2`. All common blocks are included via a single include file, which also defines some constants such as one and π in the precision currently used. I have tried hard to make switching between `real` and `double precision` as easy as possible.

The packages roughly consists of six kind of routines:

- The high-level and user-callable routines, such as `ffxd0`.
- Dotproduct calculation routines, such as `ffdot4`.
- The determinant routines, such as `ffdl4p`; the number indicates the size of the determinant and the letter the kind.
- Routines to get combinations of dilogarithms, for instance `ffcxr`; the names roughly follow the names given in [vOV90]
- Low level routines: the logarithms, dilogarithms, η functions.
- Support routines: initialisation, the error and warning system, taylor series boundaries and consistency checking.

The high-level routines first compute missing arguments such as the differences of the input parameters. Next the parameters are permuted to a position in which the evaluation is possible. All dotproducts are calculated and from these the necessary determinants are determined. In the case of the four-point function we now perform the projective transformation and compute all transformed dotproducts and differences. The determinants and dotproducts allow us to find the combinations of roots needed, which are passed on to the routines which evaluate the combinations of dilogarithms.

The most difficult part is to anticipate the cancellations among the dilogarithms without actually calculating them. This is usually done by comparing the arguments mapped to the unit circle c_i', with a safety margin. Unfortunately the choices made are not always the best, especially on the higher levels (complete C_0's or S_i's). This is the reason the user can influence the possibilities considered with the flags `l4also` and `ldc3c4`, which switch on or off the 16 dilogarithm algorithm and the expanded difference between two three-point functions.

The dilogarithms are evaluated in `ffxli2` and `ffzli2`. These expect their arguments to lie in the region $|z| < 1, \Re(z) < 1/2$ already, more general functions (used for testing) are `ffzxdl` and `ffzzdl`. The algorithm used is the expansion in $\log(1 - z)$ described in

[tHV79]. As the precision of the computer is unknown in advance fancy Chebychev polynomials and the like are not used.

The values of the logarithms and dilogarithms are placed in a big array which is only summed at the last moment. This is done to prevent false alarms of the warning system. *Every single addition* in the whole program of which one cannot prove that both operands have the same sign is checked for numerical problems with a line like

```
sum = x + y + z
xmax = max(abs(x),abs(y))
if ( abs(sum) .lt. xloss*xmax ) call ffwarn(n,ier,sum,xmax)
```

with `xloss` set to 1/8 by `ffinit`. A theoretically better way would be to compare the result to the partial sums. We are however only interested in the order of magnitude of the cancellation, and for that this method suffices.

The only other place where one can loose significant precision is in taking the logarithm of a number close to 1. All calls to the logarithm are checked by a wrapper routine for this case. A routine `dfflo1/zfflo1` is provided to evaluate $\log(1-x)$.

Finally a word on the the determinant routines. They use in general a very simplistic algorithm to find the linearly independent combination of vectors which gives the most accurate answer: try until it works. All sets are tried in order until the sum in no smaller than `xloss` times the largest term. In the larger determinants this set is remembered and tried first the next time the routine is called.

A.2.2 Timings

In table A.1 we give the timings of the scalar n-pint functions on different machines. The numbers given can only be an indication as the path taken varies wildly with the complexity of the problem. A numerical unstable set of parameters might mean much more time spent in the determinant routines and a bit less in the dilogarithms for instance. The flag `ltest` was turned off for these tests.

For a D_0, approximately 10% of the time is spent in the dilogarithms, 50% in the determinants and the rest in the sorting out and summing.

machine	B_0	C_0	D_0	E_0
NP1	0.2 ms	4.5 ms	13 ms	65 ms
Sun4	0.9 ms	8.1 ms	20 ms	90 ms
Apollo 10020	0.08 ms	1.5 ms	4.9 ms	24 ms
Atari ST	40 ms	400 ms	900 ms	5800 ms

Table A.1: Timings of the scalar n-point functions.

A.2.3 Tests

The B_0 has been tested against FormF over all parameter space, the C_0 for some 100 physical configurations and the D_0 for about 30. The E_0 is as yet untested (except for internal consistency). The only differences were in very low t-channel configurations and I have reason to distrust FormF. The limit is not approached smoothly, and very extreme kinematical configurations such as those occurring in the ZEUS luminosity monitor [vdH90] often give a `DMPX`. FF approaches the theoretically correct limit smoothly.

A.3 Installation

In this section the installation of the FF routines on a computer is discussed. We will first discuss the problems which may be caused by the Fortran used. Next the use of data files is discussed.

The routines have been written in standard (ANSI) Fortran-77, with a few extensions, which most compilers allow. The package compiles without changes on the Gould/Encore (fort), Apollo/SR10 (ftn), Meiko (mf77) and VAX (fortran/g_float). Changes are necessary for the Apollo/SR9 (ftn), Sun (f77), CDC (ftn5), Atari ST (Absoft) and possibly other compilers.

The extensions used are:

- the use of tabs.
- the use of lower case letters.
- the use of `implicit none`.

- the use of the `include` directive to include the file 'ff.h', which contains parameters and common blocks used throughout the package.
- the use of `DOUBLE COMPLEX` data type. In principle FF can also run in single precision, but the loss of 3-5 digits can often not be avoided in the evaluation of an n-point function. This may leave too little information.

All these extensions can easily be removed with a good editor. The following commands will convert the source to ANSI Fortran. (The syntax is that of the editor STEDI).

```
mark
/include 'ff.h'/
deleteline
read ff.h
/implicit none/=/implicit logical (a-z)/
/DBLE(/=/REAL(/
/DIMAG/=/AIMAG/
/DCMPLX/=/CMPLX/
/DOUBLE COMPLEX/=/COMPLEX/
end
# convert to uppercase
ctrl-u
# expand the tabs
te
```

Note that all names that have to be converted when switching from single to double precision are in capitals. It is possible to run the package in double precision real and single precision complex (the error reporting system might underestimate the accuracy in this case). To convert to single precision real (for instance on a CDC) use

```
/DOUBLE PRECISION/=/REAL/
```

It may be necessary to convert to systems with other names for the double precision complex data types and functions (e.g. IBM). The double complex functions to be transformed are `zfflo1`, `zfflog` and `zxfflg`. They are now declared as `DOUBLE COMPLEX function(args)`, change this to `COMPLEX function*16(args)`.

Generic names for the intrinsic functions `sqrt`, `log` and `log10` are used everywhere, so these need not be changed.

Note that all subroutines have names starting with `ff`, the functions have the `ff` in the middle of the name. It is hoped that this naming convention will minimise conflicts with user-defined names. The author is aware of the possible conflict with the Cern-library package 'ffread', but could not think up another key.

The FF package uses three data files: `fferr.dat`, `ffwarn.dat` and `ffperm5.dat`. The mechanism for locating these is very simple: in the subroutine which reads these files (`ffopen` and `ffwarn` in the file `ffinit`) the variable `fullname` is defined. You will have to fill in here a directory (readable by everyone using the routines) that contains the datafiles*.

A.4 Initialisation

When using the FF routines a few initialisations have to be performed in the program that calls these routines.

The common blocks used are all listed in the file 'ff.h'. If your system does not automatically save common blocks (like Absoft Fortran) it is easiest to include this file in the main program.

Furthermore, before any of the subroutines are called, a call must be made to `ffinit` to initialise some arrays of Taylor series coefficients. This routine also tries to establish the machine precision and range, causing two underflows. If this is a problem (e.g. with Gould dbx), edit this routine to a hardwired range. Finally it sets up reasonable defaults for the tracing flags (these are listed in A.5.3). This call is made automatically if one uses the `npoin` entry point.

A call to `ffexit` will check the integrity of these arrays and give a summary of the errors and warnings encountered.

Finally, on systems on which error trapping is possible it may be advantageous to use a call

```
call qsetrec(ffrcvr)
```

This forwards any floating point errors to the error reporting system. The routine qsetrec is available in the Cern library.

*for VAX/VMS one has to add the non-standard `READONLY` to the open statement

A.5 The error reporting system

A.5.1 Overview

One of the goals of this package was to give *reliable* answers. For this purpose a rather elaborate error reporting system has been built in. First, there are a few flags which govern the level of internal checking. Secondly, a count of the number of digits lost in numerical cancellations above some acceptable number (this number is defined for each function in section A.6) is default returned with any result. This count is quite conservative. *Do not forget the few digits normal everyday loss* on top of the reported losses, however: the 'acceptable' loss. Finally, a message can be given to the user where the error or warning occurred. For this to be useful, the user has to update some variables.

A.5.2 Using the system

Errors

A distinction is made between errors and warnings. An error is an internal inconsistency or a floating point error (if trapped). If an error occurs a message is printed on standard output like this (the output is truncated to fit on the page)

```
id nr     41/     7, event nr     16
error nr    32: nffeta: error:   eta is not defined for real ...
```

The first part of the id must be defined by the user. It is given by the variable `id` in the common block `/ffflags/`. I tend to use '41' for the first four-point function, '42' for the second one, etc:

```
id = 41
call ffxd0(cd0,xpi1,ier)
id = 42
call ffxd0(cd0,xpi2,ier)
```

The second part (`idsub`) is maintained internally to pinpoint the error. The event number is assumed to be `nevent` in the same common block. It too has to be incremented by the user. The error number is used internally to fetch the message text from the file

`fferr.dat`, which also includes the name of the routine in which the error occurred. If an error has occurred the variable `ier` is incremented by 100.

A call to `fferr` with the error number 999 causes a list of all errors so far to be printed out and this list to be cleared. This is used by `ffexit`.

Warnings

A warning is a loss of precision because of numerical cancellations. Only losses greater than a certain default value are noticed. This is controlled by the variable `xloss` in the common block `/ffprec/`, which is set to 1/8 by `ffinit`. A power of 2 is highly recommended. If a loss of precision greater than this tolerable, everyday loss occurs the subroutine `ffwarn` is called. The default action is to only increment the variable `ier` by the number of digits lost over the standard tolerated loss of `xloss`. Nothing is printed, but all calls occurring with the same value of the event counter `nevent` are remembered. This queue is printed when `ffwarn` is called with error number 998.

The reason for this is simply that I do not like hundreds of meaningless warnings to clutter the important ones in a big Monte Carlo. I therefore include a line like

```
if ( ier .gt. 10 ) call ffwarn(998,ier,x0,x0)
```

at the end of the calculation of one event, causing the system to report only those errors which led to a fatal loss of precision. The warning messages produced are similar to an error message:

```
id nr     41/     4, event nr   2265
warning nr    138: ffdl3p: warning: cancellations in \delta_{...
     (lost    1 digits)
```

The number of digits lost gives the number of digits which have become unreliable in the answer due to this step *over the normal loss of* `xloss`.

Another special error number is 999: this causes a list of all warnings which have occurred up to that point to be printed out plus the maximum loss suffered at that point. The routine `ffexit` uses this.

There is one warning message which does not increase `ier`: the remark that there are cancellations among the input parameters. This is the responsibility of the user. Most

routines have an alternative entry point with the differences of the parameters required as input.

The user can edit the routines `ffwarn` and `fferr` (in the file `ffinit`) to customise the error and warning reporting.

A.5.3 Debugging possibilities

There are a few flags to control the package in great detail. These are contained in the common block `/ffflags/`. The first one, `lwrite`, if on, gives a detailed account of all steps taken to arrive at the answer. This gives roughly 1000 lines of output for a four-point function. It is turned off by `ffinit`. The second one, `ltest`, turns on a lot of internal consistency checking. If something is found wrong a message like

```
ffdot4: error: dotproducts with p(10) wrong: -1795. ... -9.5E-12
```

is given. The last number gives the deviation from the expected result, in this case a relative precision of 10^{-15} was found instead of the expected 10^{-16}. The `ier` counter is *not* changed, as these are usually rounding off errors. Please report any serious errors. This flag is turned on by `ffinit`, turn it off manually once you are convinced that your corner of parameter space does not present any problems.

The next two flags, `l4also` and `ldc3c4`, control the checking of some extra algorithms. This takes time and may even lead to worse results in some rare cases. If you are pressed for speed, try running with these flags off and only switch them on when you get the warning message "`Cancellations in final adding up`". If you get mysterious warnings with the flags on, try turning them off.

Another flag for internal use, `lmem` controls a rudimentary memory mechanism which is mainly used when trying different permutations of the parameters of the three- and four-point functions. Its use is taken care of by the system.

Next there is the possibility to save the array of dotproducts used by the three and four-point function. These arrays are used by the tensor integrals.

Finally there is the possibility to to turn off all warning reporting by setting `lwarn` to `.FALSE.`. Do not do this until you are completely satisfied that there are no problems left! It will also invalidate the value of `ier`, so you will have no warning whatsoever if something goes horribly wrong.

It may be advantageous to change the flags to parameters and recompile for extra speed and smaller size. Approximately half the code of the package is for debugging purposes.

A.5.4 Summary

The following sequence has been found to be very convenient.

a) Make sure that the system can find `fferr.dat` and `ffwarn.dat` and that the routine `ffinit` is called.

b) Do a pilot run with `ltest` on to check for internal problems within the FF routines. One can also look for the best permutation of the input parameters at this stage. Please report anything irregular.

c) Run a full Monte Carlo with `ltest` off, but `lwarn` still on to check for numerical problems.

d) Only if there are *no* numerical problems left, you can turn off `lwarn` to gain the last percents in speed.

A.6 Scalar n-point functions

In general there are two routines for almost every task: one for the case that all parameters are real and one to use if one or more are complex. Infra-red divergent diagrams are calculated with a user-defined cutoff on the divergent logarithms. Planned extensions are

- the derivative of B0,
- fast special cases,
- six-point functions.

Please note that there is also an entry-point `npoin` which returns the scalar integrals plus the supported tensor integrals in a form compatible with FormF. The number of digits lost cannot be included this way, however. It is provided on request to allow old code which used FormF to run without a CDC.

A.6.1 One-point function

The one-point function `ca0` $= A_0(m^2) = \frac{1}{i\pi^2}\int d^nQ/(Q^2 - m^2)$ is calculated with the subroutines

```
subroutine ffca0(ca0,d0,xmm,cm,ier)
integer ier
DOUBLE COMPLEX ca0,cm
DOUBLE PRECISION d0,xmm

subroutine ffxa0(ca0,d0,xmm,xm,ier)
integer ier
DOUBLE COMPLEX ca0
DOUBLE PRECISION d0,xmm,xm
```

with `d0` $= \Delta = -2/\epsilon - \gamma + \log(4\pi)$ the infinity from the renormalisation scheme and the mass `xmm` $= \mu$ arbitrary. The final result should not depend on it. `xm` $= m^2$ is the internal mass *squared*. This is of course a trivial function.

A.6.2 Two-point function

Calling sequence

The two-point function `cb0` $= B_0(m_a^2, m_b^2, k^2)$ is calculated in the subroutines

```
subroutine ffcb0(cb0,d0,xmu,ck,cma,cmb,ier)
integer ier
DOUBLE COMPLEX cb0,ck,cma,cmb
DOUBLE PRECISION xmu,d0

subroutine ffxb0(cb0,d0,xmu,xk,xma,xmb,ier)
integer ier
DOUBLE COMPLEX cb0
DOUBLE PRECISION d0,xmu,xk,xma,xmb
```

with `d0` and `xmm` as in the one-point function. `xk` $= k^2$ in Bjørken and Drell metric $(+---)$ and `xma,b` $= m_{a,b}^2$ are the internal masses *squared*.

Comments

The maximum loss of precision without warning in the scalar two-point function is (`xloss`)^3 in the basic calculation plus `xloss` when adding the renormalisation terms. Numerical instabilities only occur very close to threshold ($k^2 \approx (m_a + m_b)^2$). The function can run into underflow problems if both $|m_a - m_b| \ll m_a$ and $|k^2| \ll m_a^2$. Note that this function uses Pauli metric $(+++-)$ internally.

A.6.3 Three-point function

Calling sequence

The three-point function `cc0` $= C_0(m_1^2, m_2^2, m_3^2, p_1^2, p_2^2, p_3^2)$ is calculated in the subroutines

```
subroutine ffcc0(cc0,cpi,ier)
integer ier
DOUBLE COMPLEX cc0,cpi(6)

subroutine ffxc0(cc0,xpi,ier)
integer ier
DOUBLE COMPLEX cc0
DOUBLE PRECISION xpi(6)
```

The array `xpi` should contain the internal masses squared in positions 1–3 and the external momenta squared in 4–6. The momentum `xpi(4)` $= p_1^2$ is the one between `xpi(1)` $= m_1^2$ and `xpi(2)` $= m_2^2$, and so on cyclically. The routine rotates the diagram to the best position, so only the swap $m_1^2 \leftrightarrow m_3^2$, $p_1^2 \leftrightarrow p_2^2$ can be used to test the accuracy.

There is an alternative entry point which can be used if there are significant cancellations among the input parameters.

```
subroutine ffxc0a(cc0,xpi,dpipj,ier)
integer ier
DOUBLE COMPLEX cc0
DOUBLE PRECISION xpi(6),dpipj(6,6)
```

All differences between input parameters should be given in `dpipj(i,j) = xpi(i) - xpi(j)`.

In the testing stages one can use

```
subroutine ffcc0r(cc0,cpi,ier)
integer ier
DOUBLE COMPLEX cc0,cpi(6)

subroutine ffxc0r(cc0,xpi,ier)
integer ier
DOUBLE COMPLEX cc0
DOUBLE PRECISION xpi(6)
```

It tries 2 different permutations of the input parameters and the two different signs of the root in the transformation and takes the best one. This permutation can later be chosen directly in the code.

If the requested three-point function is infra-red divergent (i.e. one internal mass 0 and the other two on-shell) the terms $\log(\lambda^2)$, with λ the regulator mass, are replaced by $\log(\delta)$. In all other terms the limit $\lambda \to 0$ is taken. The value of the cutoff parameter `delta` $= \delta$ should be provided via the common block `/ffcut/`, in which it is the first (and only) variable. This infra-red option does not yet work in case some of the masses have a finite imaginary part.

Comments

The maximum loss of precision without warning is $(\texttt{xloss})^5$. Numerical instabilities again occur very close to thresholds ($p_i^2 \approx (m_i + m_{i+1})^2$). There are discrepancies with FormF for t-channel diagrams in case $t \to 0$, but there are good reasons to distrust FormF there (the limit is not approached smoothly).

The Z vertex correction to an $ee\gamma$ vertex with one of the electrons slightly off-shell is stable only for one mirror image.

A.6.4 Four-point function

Calling sequence

`cd0` $= D_0(m_1^2, m_2^2, m_3^2, m_4^2, p_1^2, p_2^2, p_3^2, p_4^2, (p_1 + p_2)^2, (p_2 + p_3)^2)$, the four-point function, is calculated in the subroutine

```
subroutine ffxd0(cd0,xpi,ier)
integer ier
DOUBLE COMPLEX cd0
DOUBLE PRECISION xpi(13)
```

The array `xpi` should contain the internal masses squared in positions 1–4, the external momenta squared in 5–8 and $s = (p_1 + p_2)^2$, $t = (p_2 + p_3)^2$ in 9–10. Positions 11–13 should contain either 0 or

```
xpi(11) = u = +xpi(5)+xpi(6)+xpi(7)+xpi(8)-xpi(9)-xpi(10)
xpi(12) = v = -xpi(5)+xpi(6)-xpi(7)+xpi(8)+xpi(9)+xpi(10)
xpi(13) = w = +xpi(5)-xpi(6)+xpi(7)-xpi(8)+xpi(9)+xpi(10)
```

Unfortunately the complex four-point function does not yet exist in a usable form.

There are two alternative entry points. The first one can be used if there are significant cancellations among the input parameters.

```
subroutine ffxd0a(cd0,xpi,dpipj,ier)
integer ier
DOUBLE COMPLEX cd0
DOUBLE PRECISION xpi(13),dpipj(10,13)
```

in which these last elements are required and all differences between the input parameters are given in `dpipj(i,j) = xpi(i) - xpi(j)`.

The second one can be used in the testing stages.

```
subroutine ffxd0r(cd0,xpi,ier)
integer ier
DOUBLE COMPLEX cd0
DOUBLE PRECISION xpi(13)
```

It tries 6 different permutations of the input parameters and the two different signs of the root in the transformation and takes the best one. This permutation can later be chosen directly in the code.

If the requested four-point function is infra-red divergent (i.e. one internal mass 0 and the adjoining lines on-shell) the terms $\log(\lambda^2)$, with λ the regulator mass, are replaced by $\log(\delta)$. In all other terms the limit $\lambda \to 0$ is taken. The numerical value of `delta` $= \delta$ should be placed in a common block `/ffcut/`. *Due to problems in the transformation at this moment at most one propagator can have zero mass.*

Comments

The maximum loss of precision without warning is (`xloss`)[7]. There may be problems with diagrams with masses and/or momenta squared exactly zero. If you get a division by zero or the like try with a small non-zero mass.

The following diagrams are known not give an accurate answer:

a) Again, any configuration with an external momentum very close to threshold.

b) $\gamma\gamma \to \gamma\gamma$ for $s \ll m^2$

A.6.5 Five-point function

Calling sequence

The five-point function `ce0` $= E_0(m_i^2, p_i^2, (p_i + p_{i+1})^2, i = 1, 5)$ and the five four-point functions which one obtains by removing one internal leg are calculated in the subroutine

```
subroutine ffxe0(ce0,cd0i,xpi,ier)
integer ier
DOUBLE COMPLEX ce0,cd0i(5)
DOUBLE PRECISION xpi(20)
```

The array `xpi` should contain the internal masses squared in positions 1–5, the external momenta squared in 6–10 and the sum of two adjacent external momenta squared in 11–15 (the analogons of s and t in the four-point function). Positions 16–20 should contain either 0 or $(p_i + p_{i+2})^2$ (the analogon of u).

There are two alternative entry points. The first one can be used if there are significant cancellations among the input parameters.

```
subroutine ffxe0a(ce0,cd0i,xpi,dpipj,ier)
integer ier
DOUBLE COMPLEX ce0,cd0i(5)
DOUBLE PRECISION xpi(20),dpipj(15,20)
```

in which these last elements are required and all differences between the input parameters are given in `dpipj(i,j) = xpi(i) - xpi(j)`.

The second one can be used in the testing stages.

```
subroutine ffxe0r(ce0,cd0i,xpi,ier)
integer ier
DOUBLE COMPLEX ce0,cd0i(5)
DOUBLE PRECISION xpi(20)
```

It tries the 12 different permutations of the input parameters and the two different signs of the root in the transformation and takes the best one. This permutation can later be chosen directly in the code.

Comments

The five-point function has not yet been adequately tested.

The maximum loss of precision without warning is $(\texttt{xloss})^7$. There may be problems with diagrams with masses and/or momenta squared exactly zero. If you get a division by zero or the like try with a small non-zero mass.

A.7 Tensor integrals

At this moment only the vector two, three and four-point functions are available, of which the two-point functions is very badly implemented. These tensor integrals are scheme-independent, the higher order functions differ between the Passarino-Veltman scheme [PaV79] and the kinematical determinant scheme described in [vOV90].

A.7.1 Vector integrals

Two-point function

The vector two-point function $B_1 p^\mu = \int d^n Q^\mu / ((Q^2 - m_1^2)((Q+p)^2 - m_2^2)$ is calculated in

```
subroutine ffxb1(cb1,cb0,ca0i,xp,xm1,xm2,ier)
integer ier
DOUBLE PRECISION xp,xm1,xm2
COMPLEX cb1,cb0,ca0i(2)
```

The input parameters are `cb0` $= B_0$ the scalar two-point function, `ca0i(i)` $= A_0(m_i^2)$ the scalar one-point functions and the rest as in `ffxb0`. *This function must/will be improved*.

Three-point function

The subroutine for the evaluation of the vector three-point function $C_{11}p_1^\mu + C_{12}p_2^\mu = \int d^nQ^\mu/(Q^2 - m_1^2)((Q+p_1)^2 - m_2^2)((Q+p_1+p_2)^2 - m_3^2)$ is

```
subroutine ffxc1(cc1i,cc0,cb0i,xpi,piDpj,del2,ier)
integer ier
DOUBLE PRECISION xpi(6),piDpj(6,6),del2
COMPLEX cc1i(2),cc0,cb0i(3)
```

The required input parameters are `cc0` $= C_0$ the scalar three-point function, `cb0i(i)` the two-point functions with m_i^2 *missing*: `cb0i(1)` $= B_0(p_2^2, m_2^2, m_3^2)$. Further `xpi` are the masses as in `ffxc0` and `piDpj`, `del2` the dotproducts and kinematical determinant as saved by `ffxc0` when `ldot` is `.TRUE.`

Four-point function

The calling sequence for the vector four-point function `cd1i` which returns D_{11}, D_{12}, D_{13}, the coefficients of p_1^μ, p_2^μ and p_3^μ is

```
subroutine ffxd1(cd1i,cd0,cc0i,xpi,piDpj,del3,del2i,ier)
integer ier
DOUBLE PRECISION xpi(13),piDpj(10,10),del3,del2i(4)
COMPLEX cd1i(3),cd0,cc0i(4)
```

The input parameters are as follows. `cd0` $= D_0$ is the scalar four-point function, `cc0i(i)` $= C_0$(without m_i) the scalar three-point functions, `xpi` the masses as in `ffxd0` and `piDpj`, `del3` and `del2i` the dotproducts and kinematical determinant as saved by `ffxd0` and `ffxc0` when `ldot` is `.TRUE.`

A.8 Determinants

A knowledge of a few of the determinant routines may be useful to the user as well. On the one hand they can be used in other parts of the calculation, e.g. in the reduction

to scalar integrals, but they also are the place where the numerical instabilities have been concentrated. It is often useful or even necessary to import the required determinants directly from the kinematics section. We therefore list all the routines calculating determinants of external vectors and some containing internal vectors.

A.8.1 2×2 determinants

To calculate the 2×2 determinant `del2` $= \delta^{p_{i_1} p_{i_2}}_{p_{i_1} p_{i_2}}$, $p_3 = -(p_1 + p_2)$, given the dotproducts use

```
subroutine ffcel2(del2,piDpj,ns,i1,i2,i3,lerr,ier)
integer ns,i1,i2,i3,lerr,ier
DOUBLE COMPLEX del2,piDpj(ns,ns)

subroutine ffdel2(del2,piDpj,ns,i1,i2,i3,lerr,ier)
integer ns,i1,i2,i3,lerr,ier
DOUBLE PRECISION del2,piDpj(ns,ns)
```

In this `piDpj(i,j)` $= p_i \cdot p_j$ is the dotproduct of vectors p_i and p_j, `i1,i2,i3` give the position of the three vectors of which the determinant has to be calculated in this array. `lerr` should be 1.

If the dotproducts are not known there is a routine for `xlambd` $= \lambda(a_1, a_2, a_3)$, which is -2 times the determinant if `ai` $= p_i^2$.

```
subroutine ffclmb(clambd,cc1,cc2,cc3,cc12,cc13,cc23,ier)
integer ier
DOUBLE COMPLEX clambd,cc1,cc2,cc3,cc12,cc13,cc23

subroutine ffxlmb(xlambd,a1,a2,a3,a12,a13,a23,ier)
integer ier
DOUBLE PRECISION xlambd,a1,a2,a3,a12,a13,a23
```

The `aij = ai - aj` are again differences of the parameters in these routines.

An arbitrary 2×2 determinant $\delta^{p_{i_1} p_{i_2}}_{p_{j_1} p_{j_2}}$ can be obtained from `ffdl2i`:

```
subroutine ffdl2i(dl2i,piDpj,ns,i1,i2,i3,isn,j1,j2,j3,
```

```
+           jsn,ier)
   integer ns,i1,i2,i3,isn,j1,j2,j3,jsn,ier
   DOUBLE PRECISION dl2i,piDpj(ns,ns)
```

Here the vector $p_{i_3} = \texttt{isn}(p_{i_1} + p_{i_2})$ and analogously for j. (Note that the sign is important here).

If there is no connection between the two vectors one should use

```
subroutine ffdl2t(dlps,piDpj,i,j,k,l,lk,islk,iss,ns,ier)
integer in,jn,ip1,kn,ln,lkn,islk,iss,ns,ier
DOUBLE PRECISION dlps,piDpj(ns,ns)
```

to calculate $\delta^{p_i p_j}_{p_k p_l}$ with $p_{lk} = \texttt{islk}(\texttt{iss}p_l - pk)$ and no relationship between p_i, p_j assumed.

A.8.2 3×3 determinants

To calculate the 3×3 determinant $\texttt{dl3p} = \delta^{p_{i_1} p_{i_2} p_{i_3}}_{p_{i_1} p_{i_2} p_{i_3}}$ given the dotproducts `piDpj`, one can use

```
subroutine ffdl3p(dl3p,piDpj,ns,ii,ier)
integer ns,ii(6),ier
DOUBLE PRECISION dl3p,piDpj(ns,ns)
```

The array `ii(j)` gives the position of the vectors of the determinant has to be calculated in this array. We assume that $p_{ii(4)} = -p_{ii(1)} - p_{ii(2)} - p_{ii(3)}$, $p_{ii(5)} = p_{ii(1)} + p_{ii(1)}$ and $p_{ii(6)} = p_{ii(2)} + p_{ii(3)}$, with all vectors incoming.

The 3×3 determinant $\texttt{dl3q} = \delta^{s_{i_1} p_{i_2} p_{i_3}}_{p_{i_1} p_{i_2} p_{i_3}}$, which occurs in expressions for tensor integrals, is calculated by

```
   subroutine ffdl3q(dl3q,piDpj,i1,i2,i3,j1,j2,j3,
+                    isn1,isn2,isn3,jsn1,jsn2,jsn3,ier)
   integer i1,i2,i3,j1,j2,j3,isn1,isn2,isn3,jsn1,jsn2,jsn3,
+          ier
   DOUBLE PRECISION dl3q,piDpj(10,10)
```

Now the only assumptions that are made are that $p_{j_n} = \texttt{jsn}_n(p_{i_n} - \texttt{isn}_n p_{i_{n+1}})$ if $\texttt{j}_n$ is unequal to zero. *This routine should still be extended.*

A.8.3 4×4 determinants

To calculate the 4×4 determinant `dl4p` $= \delta^{p_{i_1} p_{i_2} p_{i_3} p_{i_4}}_{p_{i_1} p_{i_2} p_{i_3} p_{i_4}}$ given the dotproducts `piDpj`, one can use

```
subroutine ffdl4p(dl4p,piDpj,ns,ii,ier)
integer ns,ii(10),ier
DOUBLE PRECISION dl4p,piDpj(ns,ns)
```

The array `ii(j)` gives the position of the vectors of the determinant has to be calculated in this array. We assume that $p_{ii(5)} = -p_{ii(1)} - p_{ii(2)} - p_{ii(3)} - p_{ii(4)}$, $p_{ii(n+5)} = p_{ii(n)} + p_{ii(n+11)}$, with all vectors incoming again.

References

[ChFH79] M. Chanowitz, M. Furman, and I. Hinchliffe, *Nucl. Phys.* **B159** (1979) 225.

[dACTP98] F. del Aguila, A. Culatti, R. Muñoz Tapia, and M. Pérez-Victoria, MIT-CTP-2705, UG-FT-86/98, hep-ph/9806451.

[De93] A. Denner, *Fortschr. Phys.* **41** (1993) 307.

[DeDH97] A. Denner, S. Dittmaier and T. Hahn, *Phys. Rev.* **D56** (1997) 117.

[HaP98] T. Hahn and M. Pérez-Victoria, hep-ph/9807565.

[KüBD91] J. Küblbeck, M. Böhm, and A. Denner, *Comp. Phys. Commun.* **60** (1991) 165

[PaV79] G. Passarino and M. Veltman, *Nucl. Phys.* **B160** (1979) 151.

[Si79] W. Siegel, *Phys. Lett.* **B84** (1979) 193.

[tHV72] G. 't Hooft and M. Veltman, *Nucl. Phys.* **B44** (1972) 189.

[tHV79] G. 't Hooft and M. Veltman, *Nucl. Phys.* **B153** (1979) 365.

[vdH90] M. van der Horst, Ph.D. thesis, Universiteit van Amsterdam, 1990.

[Ve91] J.A.M. Vermaseren, *Symbolic Manipulation with FORM*, CAN, Amsterdam (1991).

[vOV90] G.J. van Oldenborgh and J.A.M. Vermaseren, *Z. Phys.* **C46** (1990) 425.