LoopTools User's Guide

June 15, 2004 Thomas Hahn

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

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

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

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

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

LoopTools is available from `http://www.feynarts.de/looptools`.

FormCalc is available from `http://www.feynarts.de/formcalc`.

FeynArts is available from `http://www.feynarts.de`.

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, too. There exists a mailing list where updates will be announced. To subscribe to this list, send a mail (any text) to `hahn-looptools-subscribe@particle.uni-karlsruhe.de`.

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

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

Contents

1 *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.

1.1 Installation

To compile the package, a Fortran-77 compiler and the GNU C and C++ compilers (`gcc` and `g++`) 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.feynarts.de/looptools`) to see if there are any pre-compiled binaries available for your platform.

LoopTools comes in a compressed tar archive `LoopTools2.tar.gz`. Choose a directory for *LoopTools* – `LoopTools`, say – and execute the following commands to unpack and compile the package.

```
gunzip -c LoopTools21.tar.gz | tar xvf -
cd LoopTools-2.1
./configure
make
make install
make clean
```

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

`libooptools.a`	the *LoopTools* library
`libooptools_check.a`	the version that cross-checks the results
`LoopTools`	the *MathLink* executable for *Mathematica*

The resulting directory structure should be

`LoopTools/`	the *LoopTools* directory
`LoopTools/$HOSTTYPE/`	directory for the compiled programs and libraries
`LoopTools/include/`	directory of the include files
`LoopTools/ff/`	directory of the *FF* files
`LoopTools/fortran/`	directory of the Fortran part of *LoopTools*
`LoopTools/mma/`	directory of the *Mathematica* part of *LoopTools*

`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. To see what its value is on your system, type the following command at the shell prompt:

```
echo $HOSTTYPE
```

Attention Linux users: the *MathLink* library that is shipped with *Mathematica* versions 4 or below is not compatible to the version of system libraries most modern Linux distributions now have (that is, glibc 2.1 or higher). Unfortunately, this does not give any compilation errors, but just results in a buggy executable. You can test whether your `LoopTools` executable is working by invoking it from the command line (it is in the `LoopTools/$HOSTTYPE/bin` directory). The correct response should be

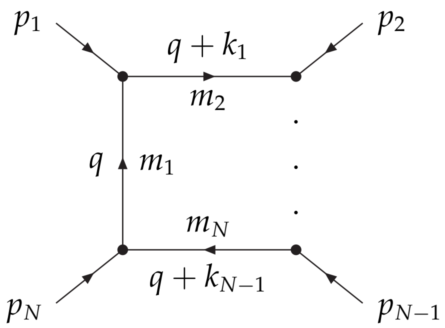
```
Create Link:
```

If instead the program crashes with a Segmentation Fault, download and install the latest *MathLink* library from `http://support.wolfram.com/mathematica/systems/linux/mathlink/libML.a` and recompile *LoopTools*.

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 `LoopTools/$HOSTTYPE`, e.g. to `/usr/local`. Superuser privileges are usually required for this.

1.2 One-Loop Integrals

Consider the following general one-loop diagram.



The integral contained in this diagram is

$$T^N_{\mu_1\ldots\mu_P} = \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2} \int \mathrm{d}^D q \, \frac{q_{\mu_1}\cdots q_{\mu_P}}{\left[q^2 - m_1^2\right]\left[(q+k_1)^2 - m_2^2\right]\cdots\left[(q+k_{N-1})^2 - m_N^2\right]} \tag{1.1}$$

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

$$\begin{aligned} p_1 &= k_1\,, & p_2 &= k_2 - k_1\,, & \ldots \quad p_N &= k_N - k_{N-1}\,, \\ k_1 &= p_1\,, & k_2 &= p_1 + p_2\,, & \ldots \quad k_N &= \sum_{i=1}^{N} p_i\,. \end{aligned} \tag{1.2}$$

The representation given in Eq. (1.1) 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.

1.2.1 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. (1.2)) [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

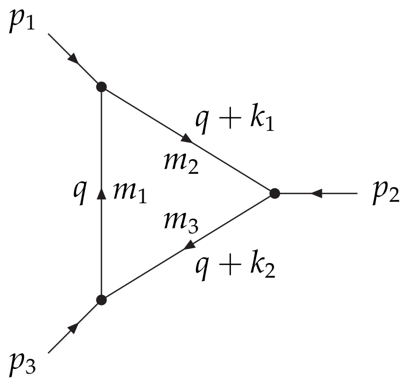
$$
\begin{aligned}
B_\mu &= k_{1\mu} B_1 \,, \\
B_{\mu\nu} &= g_{\mu\nu} B_{00} + k_{1\mu} k_{1\nu} B_{11} \,, \\
C_\mu &= k_{1\mu} C_1 + k_{2\mu} C_2 = \sum_{i=1}^{2} k_{i\mu} C_i \,, \\
C_{\mu\nu} &= g_{\mu\nu} C_{00} + \sum_{i,j=1}^{2} k_{i\mu} k_{j\nu} C_{ij} \,, \\
C_{\mu\nu\rho} &= \sum_{i=1}^{2}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

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.

1.2.2 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. (1.2)).

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*.)

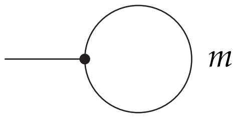
It is however important to realize that *LoopTools* functions like C_1 and C_{112} are the coefficients respectively of $k_{1\mu}$ and $k_{1\mu}k_{1\nu}k_{2\rho}$, not of $p_{1\mu}$ and $p_{1\mu}p_{1\nu}p_{2\rho}$.

1.3 Functions provided by *LoopTools*

1.3.1 One-point function

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

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

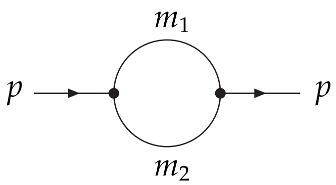


$$= \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2} \int \frac{\mathrm{d}^D q}{q^2 - m^2} \qquad \boxed{\mathtt{ms} = m^2}$$

1.3.2 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:



$$= \frac{(2\pi\mu)^{4-D}}{\mathrm{i}\pi^2} \int \frac{(\text{numerator})\,\mathrm{d}^D q}{\left[q^2 - m_1^2\right]\left[(q+p)^2 - m_2^2\right]} \qquad \boxed{\begin{aligned} \mathtt{ps} &= p^2 \\ \mathtt{m1s} &= m_1^2 \\ \mathtt{m2s} &= m_2^2 \end{aligned}}$$

1.3.3 Derivatives of Two-point functions

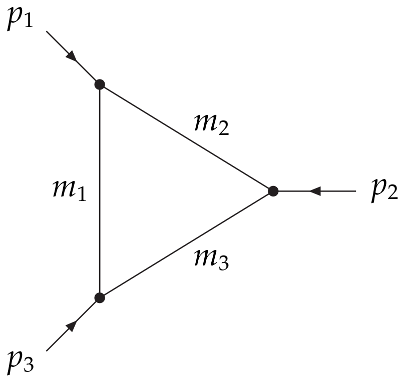
Function call	Description
`DB0(ps, m1s, m2s)`	derivative of `B0`
`DB1(ps, m1s, m2s)`	derivative of `B1`
`DB00(ps, m1s, m2s)`	derivative of `B00`
`DB11(ps, m1s, m2s)`	derivative of `B11`

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

1.3.4 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 coefficient `id`
`Cget(p1s, p2s, p1p2s, m1s, m2s, m3s)`	all three-point tensor coefficients*
	*see the Fortran, C++, and *Mathematica* section for individual calling instructions.

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`.

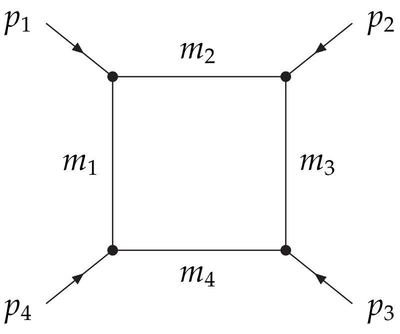
Internally, what happens when `C0i` is called is that actually *all* coefficients (with the respective arguments `p1s`, `p2s`, ...) are calculated. This is because there are a lot of intermediate results which would have to be recalculated every time the function is called for a different coefficient.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

`Cget` is the function behind `C0i` and returns all coefficients calculated for a set of parameters. Though usually the programming effort is a little higher when using this function, it is in general faster than `C0i` since only one cache lookup is needed, and not one for every coefficient.

1.3.5 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 coefficient `id`
`Dget(p1s, p2s, p3s, p4s, p1p2s, p2p3s, m1s, m2s, m3s, m4s)`	all four-point tensor coefficients† †see the Fortran, C++, and *Mathematica* section for individual calling instructions.

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



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

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

`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`. `D0i` and `Dget` work analogous to `C0i` and `Cget`, i.e. the results are also cached in memory (see end of Sect. 1.3.4).

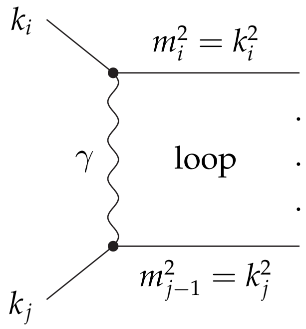
1.3.6 Ultraviolet and Infrared Divergences

The ultraviolet divergences are regularized dimensionally in *LoopTools*. The cancellation of the divergences can be checked with the two variables Δ and μ. The first one replaces the actual divergence: $\Delta = 2/(4-D) - \gamma_{\mathrm{E}} + \log 4\pi$. The second one is the dimensionful parameter introduced to keep the integral's mass dimension the same in all dimensions D (see Sect. 1.2).

The initial value for Δ is 0, the $\overline{\mathrm{MS}}$ value. Putting $\Delta = -2$ reproduces the one-loop functions of constrained differential renormalization as published in [dACTP98]. Δ is actually a redundant parameter since μ can be adjusted to have the same effect: $\mu^2_{\mathrm{new}} = \mathrm{e}^{\Delta}\mu^2_{\mathrm{old}}$.

An UV-finite result must not depend on either Δ or μ. It is hence straightforward to check UV finiteness numerically: calculate the expression with two different values for Δ (or μ, or both), and check whether the result stays the same within numerical precision. Note that μ enters logarithmically; this means that to decisively check whether an expression is really independent of μ, it must be varied on a large scale, e.g. from 1 to 10^{10}.

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



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

In *LoopTools* it is not necessary to introduce a photon mass by hand: if a requested integral is IR divergent, *LoopTools* automatically regularizes the divergence with a photon mass λ, but treats λ as an infinitesimal quantity, which means that terms of order λ or higher are discarded (i.e. only the $\log \lambda$ terms are kept).

The infinitesimal photon mass λ is a variable in *LoopTools* which can be chosen by the user. Since the final result should not depend on it after successful removal of the IR divergences, λ can be given an arbitrary numerical value despite its infinitesimal character.

To test IR finiteness numerically, one can proceed just as in the ultraviolet case: calculate the expression for two values of λ and check if the results agree. As mentioned, the λ-dependence is logarithmic, hence one has to change λ on a big scale (say from 1 to 10^{10}) to decisively check IR finiteness.

1.3.7 On Warning Messages and Checking Results

Computing reliable numeric values for the one-loop integrals is a highly non-trivial task because of possible cancellations, and requires to take into account many special cases to achieve a reasonable accuracy also in "problematic" corners of phase space. Such regions are typically thresholds and high energies.

LoopTools is built on the *FF* library which tries very hard to produce correct values. Nevertheless, it is essential to have means of cross-checking the results, particularly if such tell-tale signs of numerical problems as unsmoothness of a curve (e.g. unexpected bumps or peaks in the cross-section) are observable.

FF has a built-in warning system that checks for critical loss of accuracy. Unfortunately, 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*. *Whatever problems are still reported by FF should be considered serious!*

LoopTools instead has a much more comfortable way of cross-checking the results: If a program is linked using the linker option `-looptools_check` instead of the usual `-looptools`, every scalar function is computed once by *FF* and once using completely independent routines. Any relative deviation larger than a given threshold is reported. This threshold is 10^{-12} by default but can be changed with

```
call setmaxdev(...)
```

1.4 Using *LoopTools* with Fortran

Some technical details concerning compilation:

- Specify the location of *LoopTools* once in an environment variable (this saves a lot of typing later on). For example, in the `tcsh`, use

  ```
  setenv LT $HOME/LoopTools/$HOSTTYPE
  ```

  When compiling a program that uses *LoopTools*, add

  ```
  -I$LT/include -L$LT/lib -looptools
  ```

  to the `f77` command line. Note that in a makefile, you have to use parentheses around the environment variables, i.e. `$(LT)` instead of `$LT`.

- Fortran files that use *LoopTools* must have the extension `.F`, not `.f`. This tells the Fortran compiler that the files need to be run through the C preprocessor first. If you are using an older Fortran compiler which does not recognize the `.F` extension, use the script `F77` in the `$LT/bin` subdirectory instead of the normal `f77`.

- If you have to use f2c and cc, the script f77c in the utils subdirectory is useful for compiling Fortran programs with one command.

To use the *LoopTools* functions in a Fortran program, the file looptools.h must be included in every function or subroutine in which the *LoopTools* functions are called. Before using any *LoopTools* function, the subroutine ffini must be called. At the end of the calculation ffexi may be called to obtain a summary of errors.

A very elementary program would for instance be

```
        program simple_program
#include "looptools.h"
        call ffini
        print *, B0(1000D0, 50D0, 80D0)
        call ffexi
        end
```

Note that, as for all preprocessor commands, the # must stand at the beginning of the line. It is important to include the looptools.h via the preprocessor command #include instead of the include directive many Fortran compilers offer. This is because preprocessor variables are used in looptools.h which would otherwise not take effect. Incidentally, if you do run this program, the result should be (-4.40593283,2.7041431).

The UV- and IR-regularization parameters Δ, μ, and λ discussed in Sect. 1.3.6 can be chosen with the following subroutine calls:

```
   call setdelta(...)
   call setmudim(...)
   call setlambda(...)
```

setmudim and setlambda actually take the *squares* of the respective parameters μ and λ as arguments. Conversely, the double precision functions getdelta, getmudim, and getlambda may be used to obtain the current values of Δ, μ^2, and λ^2. The initial values are $\Delta = 0$ and $\mu = \lambda = 1$.

Whenever a 3- and 4-point tensor coefficient is requested, *LoopTools* calculates all tensor coefficients for that particular set of arguments and stores them in memory. This enormously speeds up calculations because it saves a lot of intermediate computations. However, when too many integrals are stored in the cache, the look-up time can become significant enough to wipe out the gain in speed. Particularly in large calculations it is therefore necessary to flush this internal cache from time to time. This is not done automatically since there usually exist strategic points in a program to do so. For instance, you may have a loop like

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

In general, the parameters of `C0i` and `D0i` depend on the energy, so the odds are that none of the previously calculated results can be re-used at a different energy. To discard the old results and make room for new ones use

```
call setcachelast(Ccache, 0)
call setcachelast(Dcache, 0)
```

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 Cptr, Dptr

do Energy = 100, 1000, 50
  (... calculate angle-independent integrals ...)
  Cptr = getcachelast(Ccache)
  Dptr = getcachelast(Dcache)
  do theta = cutoff, pi - cutoff, pi/180D0
    (... calculate angle-dependent integrals ...)
    call setcachelast(Ccache, Cptr)
    call setcachelast(Dcache, Dptr)
  enddo
  call setcachelast(Ccache, 0)
  call setcachelast(Dcache, 0)
enddo
```

To reduce the overhead of looking up the individual coefficients in the cache, the two functions `Cget` and `Dget` are available. They are used as in the following example:

```
        integer theC

        theC = Cget(p1, p2, p1p2, m1, m2, m3)
        c_0  = Cval(cc0, theC)
        c_1  = Cval(cc1, theC)
        c_2  = Cval(cc2, theC)
        c_00 = Cval(cc00, theC)
        (etc.)
```

which is equivalent to, but faster than,

```
        c_0  = C0i(cc0, p1, p2, p1p2, m1, m2, m3)
        c_1  = C0i(cc1, p1, p2, p1p2, m1, m2, m3)
        c_2  = C0i(cc2, p1, p2, p1p2, m1, m2, m3)
        c_00 = C0i(cc00, p1, p2, p1p2, m1, m2, m3)
        (etc.)
```

Note that pointers obtained with `Cget` or `Dget` are valid in general only as long as no call to `setcachelast` is made.

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.

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

        call ffini
        do s = 100, 1000, 50
          print *, s, "  ", SigmaH(s)
        enddo
        call ffexi
        end

        double complex function SigmaH(k2)
        double precision k2
```

```
#include "looptools.h"
        double precision MH2, MZ2, MW2, Alfa, pi, SW2
        parameter (MH2 = 100D0**2,
     +    MZ2 = 91.188D0**2,
     +    MW2 = 80.39D0**2,
     +    Alfa = 1/137.0359895D0,
     +    pi = 3.14159265358979D0,
     +    SW2 = 1 - MW2/MZ2)

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

1.4.1 Using Quadruple Precision

For most calculations, double precision is quite sufficient to yield satisfyingly accurate results. In rare cases, however, cancellations between diagrams can cause double-digit loss of precision. Since the mantissa of a double precision number has only about 15 decimal digits, the result may thus be correct only to very few digits. A case where this happens is the scattering of longitudinal gauge bosons.

Quadruple precision (16-byte real and 32-byte complex variables) have a mantissa of approximately 33 decimal digits and can cope with even the most severe cancellations. Quadruple precision does slow down the calculation, though, and is also not available on all platforms.

On platforms with a Fortran-90 compiler which supports the data type `REAL*16`, a quadruple-precision version of the *LoopTools* library can be built. This is currently the case for Alpha platforms using the Digital/Compaq compilers.

The procedure to build this library is as follows:

- change to the main *LoopTools* directory and, if you haven't done so already, run `./configure` as for the double-precision case,

- type `make clean` to remove any object files left over from the compilation of the double-precision libraries,
- change into the subdirectory `qprec` and invoke `make` there.

This will build two libraries, `libooptoolsQ.a` and `libooptoolsQ_check.a`, which are the quadruple-precision versions of `libooptools.a` and `libooptools_check.a`, respectively.

These libraries are used exactly as in the double-precision case, except that for compiling a program that uses these libraries, the script `q77` from the `qprec` directory must be used instead of `f77`.

1.5 Using *LoopTools* with C++

Some technical details:

- Like in the Fortran case, it saves a lot of typing to specify the location of *LoopTools* once in an environment variable. For example, in the `tcsh`, use

  ```
  setenv LT $HOME/LoopTools/$HOSTTYPE
  ```

  Then compile the programs that use *LoopTools* with the following command:

  ```
  $LT/bin/ccf -I$LT/include (source files) -L$LT/lib -looptools
  ```

  `ccf` is a script to compile C and C++ programs and link them with Fortran libraries, in this case `libooptools.a`. Note that in a makefile, you have to use parentheses around the environment variables, i.e. `$(LT)` instead of `$LT`.

- 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.

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

In C++, an elementary program would be

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

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

The UV- and IR-regularization parameters Δ, μ, and λ discussed in Sect. 1.3.6 can be chosen with

```
setdelta(...);
setmudim(...);
setlambda(...);
```

and retrieved with `getdelta`, `getmudim`, and `getlambda`, respectively. The initial values are $\Delta = 0$ and $\mu = \lambda = 1$. Note that `setmudim` and `setlambda` actually take the *squares* of the respective parameters μ and λ as arguments.

Flushing the internal cache for the 3- and 4-point functions from time to time is also a good idea in C++. This is done by calling

```
setcachelast(Ccache, 0);
setcachelast(Dcache, 0);
```

It is also possible to obtain the location in memory where the coefficients are stored. This helps reduce the overhead in looking up each coefficient function in the cache. The functions `Cget` and `Dget` return integers which can then be used with the macros `Cval` and `Dval` to retrieve the individual tensor coefficients. For example,

```
int theC = Cget(p1, p2, p1p2, m1, m2, m3);
c_0 = Cval(cc0, theC);
c_1 = Cval(cc1, theC);
c_2 = Cval(cc2, theC);
c_00 = Cval(cc00, theC);
(etc.)
```

is equivalent to, but faster than,

```
c_0 = C0i(cc0, p1, p2, p1p2, m1, m2, m3);
c_1 = C0i(cc1, p1, p2, p1p2, m1, m2, m3);
c_2 = C0i(cc2, p1, p2, p1p2, m1, m2, m3);
c_00 = C0i(cc00, p1, p2, p1p2, m1, m2, m3);
(etc.)
```

Note that pointers obtained with `Cget` or `Dget` are valid in general only as long as no call to `setcachelast` is made.

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 <complex.h>
#include <fstream.h>
#include "clooptools.h"

double_complex SigmaH(double);

main()
{
  double s;
```

```
  ffini();
  for(s = 100.; s <= 1000.; s += 50.)
    cout << s << "\t" << SigmaH(s) << endl;
  ffexi();
}

#define MH2 (100.*100.)
#define MZ2 (91.188*91.188)
#define MW2 (80.4*80.4)
#define Alfa (1./137.0359895)
#define pi 3.14159265358979
#define SW2 (1. - MW2/MZ2)

double_complex SigmaH(double k2)
{
  return Alfa/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 );
}
```

1.6 Using *LoopTools* with *Mathematica*

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

```
set path=($path $HOME/LoopTools/$HOSTTYPE/bin)
```

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["LoopTools"]

Out[1]= LinkObject[LoopTools, 1, 1]
```

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

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

`Cget` and `Dget` return a list of rules containing all three- and four-point tensor coefficients, respectively, e.g.

```
In[3]:= Cget[80, 80, 10000, 300, 100, 200]//InputForm

Out[3]//InputForm=
  {cc0 -> 0.0003683322958259527 - 0.00144304878124425*I,
   cc1 -> 0.0000369199114668660 7 + 0.0008063637675463306*I,
   cc2 -> -0.0002186870966525929 + 0.0003255577507551812*I,
   cc00 -> -1.468122864600498 + 0.6620214671984382*I,
   cc11 -> -0.0001383963649940767 - 0.0005211388919006447*I,
   cc12 -> 0.00005607420875500784 - 0.0001466442566605745*I,
   cc22 -> 0.0001038232033882128 - 0.0001572866825209231*I,
   cc001 -> 0.4339544374355454 - 0.1905346035793642*I,
   cc002 -> 0.5179247985708856 - 0.2390535391455292*I,
   cc111 -> 0.0001637407816195954 + 0.0003561351446381443*I,
   cc112 -> -0.00001499429891688691 + 0.00008510756809075344*I,
   cc122 -> -0.00002351641063613291 + 0.00005055502592614985*I,
   cc222 -> -0.00005956786867352272 + 0.0001019629695390 97*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, `LoopTools` can be uninstalled:

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

The UV- and IR-regularization parameters Δ, μ, and λ discussed in Sect. 1.3.6 can be chosen with

```
SetDelta[...]
SetMudim[...]
SetLambda[...]
```

and can be retrieved with `GetMudim`, `GetLambda`, and `GetDelta`, respectively. Initially $\Delta = 0$ and $\mu = \lambda = 1$. Note that `SetMudim` and `SetLambda` actually take the *squares* of the respective parameters μ and λ as arguments.

The internal caches for `C0i` and `D0i` (see Sects. 1.3.4 and 1.4) can be emptied with

```
SetCacheLast[Ccache, 0]
SetCacheLast[Dcache, 0]
```

Invoking `SetCacheLast` is usually only necessary when large numbers of three- and four-point integrals are requested.

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 $|z| < 1, \mathrm{Re}(z) < 1/2$ 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 `ffini`. 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 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 `ffini`) 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 `ffini` 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 `ffini`. 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 `ffini`) 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 `ffini`. 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 `ffini`, 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 `ffini` 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 (`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)` $= C_0(\text{without } m_i)$ 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 $p_{i_3} = \mathtt{isn}(p_{i_1}+p_{i_2})$ 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 $p_{lk} = \mathtt{islk}(\mathtt{iss}p_l - pk)$ and no relationship between p_i, p_j assumed.

A.8.2 3×3 determinants

To calculate the 3×3 determinant $\mathtt{dl3p} = \delta^{p_{i_1} p_{i_2} p_{i_3}}_{p_{i_1} p_{i_2} p_{i_3}}$ 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 $\mathtt{dl3q} = \delta^{s_{i_1} p_{i_2} p_{i_3}}_{p_{i_1} p_{i_2} p_{i_3}}$, 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 $p_{j_n} = \mathtt{jsn}_n(p_{i_n} - \mathtt{isn}_n p_{i_{n+1}})$ if $\mathtt{j}_n$ 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

[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.

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

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

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

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

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

Index