NIKHEF-H/90-15

FF

a package to evaluate one-loop Feynman diagrams

G. J. van Oldenborgh
NIKHEF-H
P.O. Box 41882
NL-1009 DB Amsterdam

24 September 1990

Abstract

A short description and a user's guide of the FF package are given. This package contains routines to evaluate numerically the scalar one-loop integrals occurring in the evaluation in one-loop Feynman diagrams. The algorithms chosen are numerically stable over most of parameter space.

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 [1], 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 [2]. 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 This costs much time and has to be extended for every new calculation, as often the limits taken are no longer valid.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 [3]. This paper describes version 1.0 which contains the following units:

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

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

- 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 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 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 6) and determinants (section 8) is given, listing parameters, loss of precision and comments. Finally some sample input and output is given in section 9.

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 [3]. 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.

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 [3]

- 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 [1]. 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 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.

2.2 Timings

In table 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.

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 1: Timings of the scalar n-point functions.

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.

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 [4] often give a `DMPX`. FF approaches the theoretically correct limit smoothly.

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[1].

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 5.3). This call is made automatically if one uses the `npoin` entry point.

A call to `ffexi` 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.

---

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

5 The error reporting system

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 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.

5.2 Using the system

5.2.1 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 `ffexi`.

5.2.2 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 `ffexi` 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.

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.

5.4 Summary

The following sequence has been found to be very convenient.

1. Make sure that the system can find `fferr.dat` and `ffwarn.dat` and that the routine `ffini` is called.

2. 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.

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

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

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.

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.

6.2 Two-point function

6.2.1 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*.

6.2.2 Comments

The maximum loss of precision without warning in the scalar two-point function is $(\texttt{xloss})^3$ 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.

6.3 Three-point function

6.3.1 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.

6.3.2 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.

6.4 Four-point function

6.4.1 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.*

6.4.2 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:

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

2. $\gamma\gamma \to \gamma\gamma$ for $s \ll m^2$

6.5 Five-point function

6.5.1 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.

6.5.2 Comments

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

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.

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 [2] and the kinematical determinant scheme described in [3].

7.1 Vector integrals

7.1.1 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.*

7.1.2 Three-point function

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

7.1.3 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.`.

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.

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 $p_{i_3} =$ `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 $p_{lk} =$ `islk`(`iss`$p_l - pk$) and no relationship between p_i, p_j assumed.

8.2 3×3 determinants

To calculate the 3×3 determinant `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 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.*

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.

9 Sample input and output

The example chosen is the same that is given with FormF, although the B_1' is not computed and set to a very large value. On my NeXTstation with f2c and gcc I get the following output.

```
$ make npointes
f77 -c npointes.f
   [ many more lines]
$ npointes
 ====================================================
   FF 2.0, a package to evaluate one-loop integrals
 written by G. J. van Oldenborgh, NIKHEF-H, Amsterdam
```

```
 ====================================================
 for the algorithms used see preprint NIKHEF-H 89/17,
 'New Algorithms for One-loop Integrals', by G.J. van
 Oldenborgh and J.A.M. Vermaseren, published in
 Zeitschrift fuer Physik C46(1990)425.
 ====================================================
 ffini: precx =   4.44089209E-16
 ffini: precc =   4.44089209E-16
 ffini: xalogm =   4.94065645E-324
 ffini: xclogm =   4.94065645E-324
   .0000000000000E+00  .9156199386460E+06
 NPOIN: warning: D4 is not yet supported
 NPOIN: warning: B1' seems also not yet supported
 -.3100623399361E+02 -.2054369006935E+03 -.5269961187649E-14 -.1218492318111E-08
  .0000000000000E+00  .1997630170032+305 -.1033542556010E+02 -.6793071440282E+02
  .0000000000000E+00  .1739461728624E-08  .0000000000000E+00 -.1286491875423E-08
  .0000000000000E+00  .4952559280190E-18  .0000000000000E+00 -.4571732002955E-18
 ALIJ: error: not implemented
   .0000000000000E+00

total number of errors and warnings
===================================
fferr: no errors
ffwarn:       1 times  18: ffxb0p: warning: cancellations in equal masses, \
                                   complex roots, can be avoided.
     (lost at most a factor    8.20    )
ffwarn:       1 times 129: zxfflg: warning: taking log of number close to 1\
                                   , must be cured.
     (lost at most a factor    9.75    )
ffwarn:      10 times 163: ffxc1:  warning: cancellations in cc1.
     (lost at most a factor    244.    )
ffwarn:       1 times 164: ffxd1:  warning: cancellations in cd1.
     (lost at most a factor    8.33    )
```

10 Distribution and error reports

The Fortran package FF can of course be freely copied and used. However, please do not change anything so that others which copied your code can be sure which version they are using. The version released in summer 1990 is version 1.0. A copy of the most recent version, 2.0 (1998) (roughly $4.4\,10^4$ lines) and this users guide can be obtained in the following ways:

- via the web at `http://www.xs4all.nl/~gjvo/FF.html`
- via anonymous ftp from `ftp.nikhef.nl`.

Please report any problems you encounter when using these routines to me. Although I am no longer working in particle physics, I still support this library. My e-mail address is `GeertJan@vanOldenborgh.net`, `t19@nikhef.nl` also still works.

11 Changes

The one major change has been to rename `ffinit` and `ffexit` to `ffini` and `ffexi`, to avoid naming conflicts with the cernlib FF tape handling package. A detailed change-log can be found in the file `README` in the distribution.

References

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

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

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

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

[5] R. P. Feynman, Phys. Rev. **76**, 769 (1949).