FormCalc 8: BETTER ALGEBRA AND VECTORIZATION*

B. Chokoufe Nejad, J.-N. Lang

Institut für Theoretische Physik und Astrophysik, Universität Würzburg
Am Hubland, 97074 Würzburg, Germany

T. Hahn, E. Mirabella

Max-Planck-Institut für Physik
Föhringer Ring 6, 80805 Munich, Germany

(Received November 4, 2013)

We present Version 8 of the Feynman-diagram calculator FormCalc. New features include, in particular, significantly improved algebraic simplification as well as vectorization of the generated code. The Cuba Library, used in FormCalc, features checkpointing to disk for all integration algorithms.

DOI:10.5506/APhysPolB.44.2231
PACS numbers: 02.70.Wz, 12.38.Bx, 02.60.Jh

1. Introduction

The Mathematica package FormCalc [1] simplifies Feynman diagrams generated with FeynArts [2] up to one-loop order. It provides the analytical results and can generate Fortran code for the numerical evaluation of the squared matrix element. Cuba is a library for multidimensional numerical integration which is included in FormCalc but can also be used independently. This note presents the following features new in FormCalc 8 and Cuba 3.2:

— Better algebraic simplification using FORM 4 features.

— Vectorization of the helicity loop.

— Automated C-code generation.

— Optimizations for unitarity methods.

— Checkpointing for all Cuba algorithms.

* Presented at the XXXVII International Conference of Theoretical Physics "Matter to the Deepest" Ustroń, Poland, September 1–6, 2013.

2. Improvements in the algebraic simplification

The algebraic simplification of Feynman amplitudes is split between Mathematica and FORM. In a preprocessing stage, Mathematica translates the elements of a FeynArts amplitude into FORM syntax and writes them to an input file for FORM. (Note that none of the FeynArts symbols are directly redefined, such that processing does not start immediately when the amplitude is generated.) FORM then does the major part of the symbolic simplification. In a postprocessing step, the FORM output is read and returned to Mathematica by FormCalc's ReadForm MathLink utility.

Many new and useful features were introduced in FORM 4 [3], most notably abbreviationing and factorization. The FORM part of FormCalc 8 has been rewritten to take advantage of these facilities, resulting in significantly improved algebraic simplification.

2.1. Abbreviationing

Once a partial expression is considered final at a particular point in the FORM program it is abbreviated, *i.e.* substituted by a symbol. This not only shortens the active expressions but makes the abbreviated parts inert, such that subsequent id-statements do not spend time on matching these, thus making the FORM code run faster.

A similar technique has been used since Version 6 [5], where the FORM expressions were sent on a round-trip to Mathematica halfway through the evaluation for introducing abbreviations. Since this involved quite some transmission overhead, it was performed only once during each FORM run. With abbreviationing built into FORM now, abbreviations are introduced whenever possible, thereby obviating the extra pass to Mathematica.

Abbreviationing also serves to prevent FORM's automatic expansion of expressions, *i.e.* it preserves a (pre)factorized structure, which is particularly useful in combination with the new factorization available in FORM (see Sect. 2.2 below).

What is more, since Mathematica receives an expression in many small pieces rather than one large chunk, more aggressive simplification functions can be applied upon return to Mathematica at a reasonable efficiency. To this end, FormCalc wraps a zoo of simplification functions around various parts of the amplitude. All of these are 'transparent' in the sense that they can be replaced by `Identity` without affecting the numerical result. The three most important ones are listed below, a complete inventory is given in the FormCalc manual.

`FormSub` is applied to subexpressions of an amplitude.

`FormDot` is applied to combinations of dot products in an amplitude.

`FormMat` is applied to the coefficients of matrix elements ('`Mat`').

On the technical side, since the abbreviations are also transmitted to Mathematica as such (*i.e.* not back-substituted into the expressions), the volume of data transferred is significantly reduced and the final expression is stored efficiently as multiple instances of a subexpression are taken care of by reference count (same as with `Share[]`).

At the moment, FormCalc does not use FORM 4's "`format O`*n*" output optimization, as it is not yet clear how to combine it with the postprocessing in ReadForm and Mathematica.

2.2. Factorization

FORM's new 'full' factorization (over the rationals) makes it possible to simplify expressions much better already inside of FORM. The old 'simple' factorization (pulling out common symbols from an expression) is still used in instances where full factorization is too expensive.

Potentially time-consuming instances of the `factarg` command in the FORM code can be suppressed by setting the `NoCostly` → `True` option of `CalcFeynAmp`. This is occasionally necessary in models with more complex couplings such as the MSSM.

Plain factorization is not a cure-all for arbitrary expressions, however. For example, while the following expression is not factorizable as a whole

```
-2*e2.k5*S35 + 2*e2.k5*T24 + 2*e2.k5*T14 - 2*e2.k5*MT2 +
2*e2.k5*S - 3*e2.k6*S35 - e2.k6*S45 - e2.k6*T25 - e2.k6*T15 +
4*e2.k6*T24 + 4*e2.k6*T14 - 4*e2.k6*MT2 + 4*e2.k6*S
```

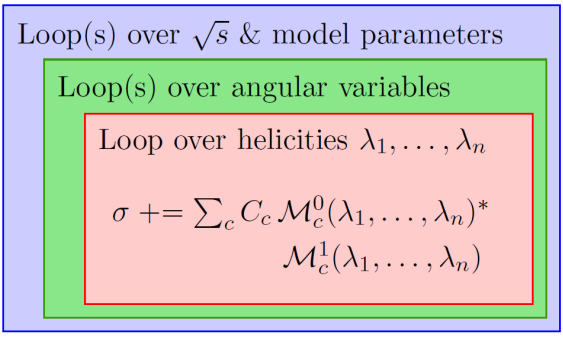
it easily admits further compactification by collecting with respect to the dot products first:

```
-2*(MT2 - S + S35 - T14 - T24)*e2.k5 -
(4*MT2 - 4*S + 3*S35 + S45 - 4*T14 + T15 - 4*T24 + T25)*e2.k6
```

FormCalc takes typical objects such as dot products into account, of course. Still, for a general expression, it is not straightforward to find a suitable simplification procedure, which is why it is useful to have functions like `FormDot` through which one can apply more sophisticated functions such as Mathematica's `Simplify`.

3. Vectorization of the helicity loop

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



The helicity loop is not only strategically the most desirable but also the most obvious candidate for concurrent execution, as FormCalc does not insert explicit helicity states during the algebraic simplification [4]. That is, the amplitude is a numerical function of the helicities λ_i and not a bunch of (different) functions for each helicity combination

$$\mathcal{M} = \mathcal{M}(\lambda_1, \lambda_2, \ldots) \neq \{\mathcal{M}_{--\ldots}, \ \mathcal{M}_{+-\ldots}, \ \mathcal{M}_{-+\ldots}, \ \mathcal{M}_{++\ldots}\} \, .$$

Such a design is known as the Single Instruction Multiple Data (SIMD) in computer science since a single code ($\mathcal{M}$) is independently run for multiple data (λ_i), and is conceptually easy to parallelize or vectorize.

Parallelization on the CPU's cores using `fork`/`wait` has been available from Version 7.5 on [7]. The drawback of this method is that it competes for compute cores in particular with Cuba. For better efficiency, the cores should be assigned to Cuba since it computes entire phase-space points in parallel, not just the helicity loop.

GPU parallelization was attempted using OpenCL but we found that it was not too efficient. Since the transfer of data between the CPU and the GPU is relatively time-consuming, we believe that the distribution of the helicity-independent variables from the CPU to the GPU outweighed the parallelization gains.

Eventually, the best speedup we could achieve was with vectorization. With Intel's x86 vector instructions, there is essentially no overhead.

3.1. Vectorization in C

Our implementation in C is based on the vector data type extensions offered by gcc and Intel's icc. Unfortunately, these are restricted to real algebra, even in C99.

Complex addition is obviously not a problem and complex multiplication might have been solved through C++'s operator overloading. Since we wanted to stick to C to avoid linking hassles with Fortran object files, we

adjusted the C-code generation in Mathematica to insert explicit macros for the multiplication of complex vectors: `SxH` stands for "scalar times helicity vector" and `HxH` for "helicity vector times helicity vector". Helicity vectors are declared with `HelType`. (To avoid confusion with the Minkowski four-vectors, we use prefixes '`Hel`' or '`H`' to denote helicity vectors in the SIMD sense.)

Depending on the hardware features indicated by preprocessor flags, these macros emit explicit SSE3 or AVX instructions. For SSE3, the maximum vector length is 1 (2 doubles per operation) which may at first not seem very useful, but besides performing addition twice as fast there exists an efficient complex multiplication routine with 2.5 instructions instead of 6. For AVX (requires i7 'Sandy Bridge' or higher), the maximum vector length is 2 (4 doubles per operation). Again, the complex multiplication can be formulated fairly efficiently using Intel's vector instructions. Overall, we found a speedup of 3.7 out of theoretical 4 with AVX for the helicity loop.

Currently the configure script does not automatically add flags to switch on SSE3 or AVX instructions, *e.g.* gcc needs the extra flag `-march=native` to enable all features of the CPU used for compilation. This may change in the future. A related question is which default to choose for executables that could potentially be run on a cluster of computers with differing SIMD capabilities.

3.2. Vectorization in Fortran

Vector data types are standard fare in Fortran 90 and so not only complex vectors are allowed but one can, in principle, choose arbitrary vector lengths. On the downside, the actual deployment of vector instructions is at the discretion of the compiler and may not be chosen for vector lengths incommensurable with the hardware.

Even though Fortran 90 is an effective requirement for vectorized computation, the code is still generated in fixed format and can be made compatible with Fortran 77 through preprocessor definitions, *e.g.* for inclusion in legacy packages.

4. Automated C-code generation

C-code generation has been available from FormCalc 7 on [6] but now its use is mostly automatic, *i.e.* also drivers and utility files are available in C. In fact, only the declarations needed to be translated as the initialization still takes place in Fortran and the C object files are simply linked in. For this to work, the layout of C's structs must match Fortran's common blocks, of course. Private declarations, *e.g.* for new models, are not automatically translated, but this is fairly straightforward as can be seen by comparing the C and Fortran versions of *e.g.* the Standard Model declarations.

To switch from Fortran to C output, the following statement needs to precede the output commands (*e.g.* `WriteSquaredME`, `WriteRenConst`):

```
SetLanguage["C"]
```

The output is by default in C99, because of complex numbers, but can easily be made to work with C++ by redefining the abstract data type `ComplexType`. Even without SIMD vectorization, and perhaps remarkably so for Fortran aficionados, C and Fortran versions of the same amplitude show very similar performance figures, *i.e.* there is no penalty for using C.

5. Optimizations for unitarity methods

FormCalc can generate amplitudes for evaluation with the OPP (Ossola, Papadopoulos, Pittau [10]) unitarity method as implemented in the two libraries CutTools [11] and Samurai [12]. Instead of introducing tensor coefficients [9], the whole numerator is placed in a subroutine, as in:

$$\begin{aligned}\varepsilon_1^\mu \varepsilon_2^\nu B_{\mu\nu}\left(p, m_1^2, m_2^2\right) &= (\varepsilon_1 \cdot \varepsilon_2) B_{00} + (\varepsilon_1 \cdot p)(\varepsilon_2 \cdot p) B_{11} \quad \text{(tensor coeff.)} \\ &= B_{\text{cut}}\left(2, N, p, m_1^2, m_2^2\right), \quad \text{(OPP)} \\ \text{where} \quad N(q_\mu) &= (\varepsilon_1 \cdot q)\,(\varepsilon_2 \cdot q)\,.\end{aligned}$$

The numerator subroutine N will be sampled by the OPP function (B_{cut} in this example). The first argument of B_{cut}, 2, refers to the maximum power of the integration momentum q in N.

The OPP procedure indeed generates significantly fewer terms than the traditional Passarino–Veltman decomposition, nevertheless a naive implementation runs quite a bit slower than its counterpart with tensor coefficients. This section describes our attempts to optimize the OPP performance. We were able to bring the slowdown from originally a factor 10 to about a factor 3 for multiplicities such as $2 \to 3$ and hope to improve matters further. To be fair, OPP was in the first place designed to increase the reach of one-loop calculations to higher-leg multiplicities and not so much to speed up the ones with not so many legs.

The major part of the slowdown (at least half of that factor 10) comes from the fact that the OPP master integrals (the scalar integrals A_0, B_0, C_0, D_0) are naively computed over and over again. This is because the OPP functions must be evaluated inside the helicity loop since the numerator subroutine depends on the helicities. The scalar integrals contain only the denominators and thus could simply be moved outside the helicity loop.

In FormCalc, we generate code that foresees a split between the computation of the masters and their use in assembling the tensor integrals, for example:

```
ComplexType mas145(Mcc)
...
call Cmas(mas145, (C0 args))
...
call Ccut(mas145, num, (C0 args))
```

The complex array `mas145` stores the master integrals computed by `Cmas` (outside the helicity loop) and used by `Ccut` (inside the helicity loop). Unfortunately, so far, none of the available OPP libraries allows this decomposition even though the two tasks '`Cmas`' and '`Ccut`' must be completed internally in some way or another already now. LoopTools alleviates the situation by retrieving recurring masters from its cache, though even here the lookup time could be eliminated with the above construction.

Some packages address this problem by moving the helicity sum into the numerator. This works if only the interference term is sought since then the amplitude contains at most one loop integration in each term

$$\sum_\lambda 2\,\mathrm{Re}\,\mathcal{M}_0^* \underbrace{\int d^4q \frac{N}{D\cdots}}_{\sim\mathcal{M}_1} = \int d^4q \frac{\sum_\lambda 2\,\mathrm{Re}\,\mathcal{M}_0^* N}{D\cdots}\,.$$

In FormCalc, we do not pursue this strategy, firstly because it is not applicable if the tree-level contribution is zero (or so small that including the loop-squared part becomes necessary) and secondly because it is not obvious how this evaluation fits into the present abbreviation concept.

Subexpressions of the numerator function (coefficients, summands, *etc.*), independent of q, are pulled out and computed once, ahead of invoking the OPP function, using FormCalc's abbreviationing machinery [13]. In particular in BSM theories, these coefficients can be lengthy such that pulling them out significantly increases performance.

Our implementation admits mixing the Passarino–Veltman decomposition with OPP in the sense that one chooses an integer n starting from which an n-point function is treated with OPP methods. For example, `OPP` $\rightarrow$ `4` means that A, B, C functions are treated with the Passarino–Veltman and D and up with OPP.

We optimize OPP calls to reduce sampling effort, *e.g.* by collecting denominators, as in

$$\frac{N_4}{D_0D_1D_2D_3} + \frac{N_3}{D_0D_1D_2} \rightarrow \frac{N_4 + D_3N_3}{D_0D_1D_2D_3}\,.$$

Depending on the number of denominators and the rank, joining integrals is not universally better. Rather, we tabulated the sampling behavior of Samurai and CutTools such that the algorithm can determine the optimal splitting.

The Ninja library implements the D-dimensional integrand reduction via Laurent expansion [14], which constructs the tensor integral from fewer samples in a numerically more stable way. Ninja requires a slightly modified numerator subroutine which is currently being implemented in FormCalc.

The profiler pointed us to a bottleneck in fermion chains: Before, we were using elementary operations to build up the fermion chain, which we have now merged into a single inlined function call

$$\begin{aligned}\langle u|\,\sigma_\mu\overline{\sigma}_\nu\sigma_\rho\,|v\rangle\,k_1^\mu k_2^\nu k_3^\rho &= \langle u|\,k_1\overline{k}_2 k_3\,|v\rangle\\ \text{(old)} &= \texttt{SxS(}u\texttt{, VxS(}k_1\texttt{, BxS(}k_2\texttt{, VxS(}k_3\texttt{, }v\texttt{))))}\\ \text{(new)} &= \texttt{ChainV3(}u, k_1, k_2, k_3, v\texttt{)}\,.\end{aligned}$$

The elementary operations could not be inlined in Fortran because they returned a 2-component spinor, not a scalar value.

The helicity information of an OPP integral's prefactor is taken into account in an extra argument. That is, if a term in the amplitude is known to become zero for a particular helicity combination due to its prefactor (because of massless external particles, say), the evaluation of the loop integral therein is cut short. For example, $\texttt{Dcut}(3, N, 1 - \texttt{Hel1}, \dots)$ is actually computed only if $\texttt{Hel1} \neq 1$.

6. Checkpointing in Cuba

Cuba's current Version 3.2 allows checkpointing for all routines. Checkpointing means writing out the integrator's complete state to disk to be able to recover from the last state after a crash. In a long-running calculation, this may mean losing one hour instead of one day.

Checkpointing is enabled by specifying a name for the state file. Note that, since only Vegas had this functionality in Version 3.0, the invocation of the other routines has changed to incorporate the extra state file argument. We always write the state to a new file and remove the former state file only when the new one is successfully stored. This makes checkpointing fail-safe since even a crash during saving is recoverable. If the integration finishes successfully, the state file is removed.

The checkpoints have been implemented in the serial regions of the code which ensures reliable behavior regardless of parallelization.

Version 3.2 furthermore relaxes several restrictions on the compiler, it is now fully C99-compliant and uses no gcc extensions.

7. Summary

FormCalc 8 (`http://feynarts.de/formcalc`) has many new and improved features, most notably better algebra, a vectorized helicity loop, and OPP improvements. The Cuba library (`http://feynarts.de/cuba`), also included in FormCalc, adds checkpointing for all four integration algorithms, which is useful for resuming interrupted long-running integrations.

REFERENCES

[1] T. Hahn, M. Pérez-Victoria, *Comput. Phys. Commun.* **118**, 153 (1999) [arXiv:hep-ph/9807565].

[2] T. Hahn, *Comput. Phys. Commun.* **140**, 418 (2001) [arXiv:hep-ph/0012260].

[3] J. Kuipers, T. Ueda, J.A.M. Vermaseren, J. Vollinga, *Comput. Phys. Commun.* **184**, 1453 (2013) [arXiv:1203.6543 [cs.SC]].

[4] T. Hahn, *Nucl. Phys. Proc. Suppl.* **116**, 363 (2003) [arXiv:hep-ph/0210220].

[5] T. Hahn, *PoS* **ACAT2008**, 121 (2008) [arXiv:0901.1528 [hep-ph]].

[6] S. Agrawal, T. Hahn, E. Mirabella, *J. Phys. Conf. Ser.* **368**, 012054 (2012) [arXiv:1112.0124 [hep-ph]].

[7] S. Agrawal, T. Hahn, E. Mirabella, *PoS* **LL 2012**, 046 (2012) [arXiv:1210.2628 [hep-ph]].

[8] T. Hahn, *Comput. Phys. Commun.* **168**, 78 (2005) [arXiv:hep-ph/0404043].

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

[10] G. Ossola, C. Papadopoulos, R. Pittau, *Nucl. Phys.* **B763**, 147 (2007) [arXiv:hep-ph/0609007].

[11] G. Ossola, C. Papadopoulos, R. Pittau, *J. High Energy Phys.* **0803**, 042 (2008) [arXiv:0711.3596 [hep-ph]].

[12] P. Mastrolia, G. Ossola, T. Reiter, F. Tramontano, *J. High Energy Phys.* **1008**, 080 (2010) [arXiv:1006.0710 [hep-ph]].

[13] T. Hahn, *PoS* **ACAT2010**, 078 (2010) [arXiv:1006.2231 [hep-ph]].

[14] P. Mastrolia, E. Mirabella, T. Peraro, *J. High Energy Phys.* **1206**, 095 (2012) [*Erratum ibid.* **1211**, 128 (2012)] [arXiv:1203.0291 [hep-ph]].