FormCalc 8

Better Algebra and Vectorization

Thomas Hahn

Max-Planck-Institut für Physik
München

What others have to say about Version 8:

“More beautiful, more flexible, more you.”

– from microsoft.com/en-us/windows8/meet.

Diagram Evaluation in FeynArts, FormCalc, LoopTools

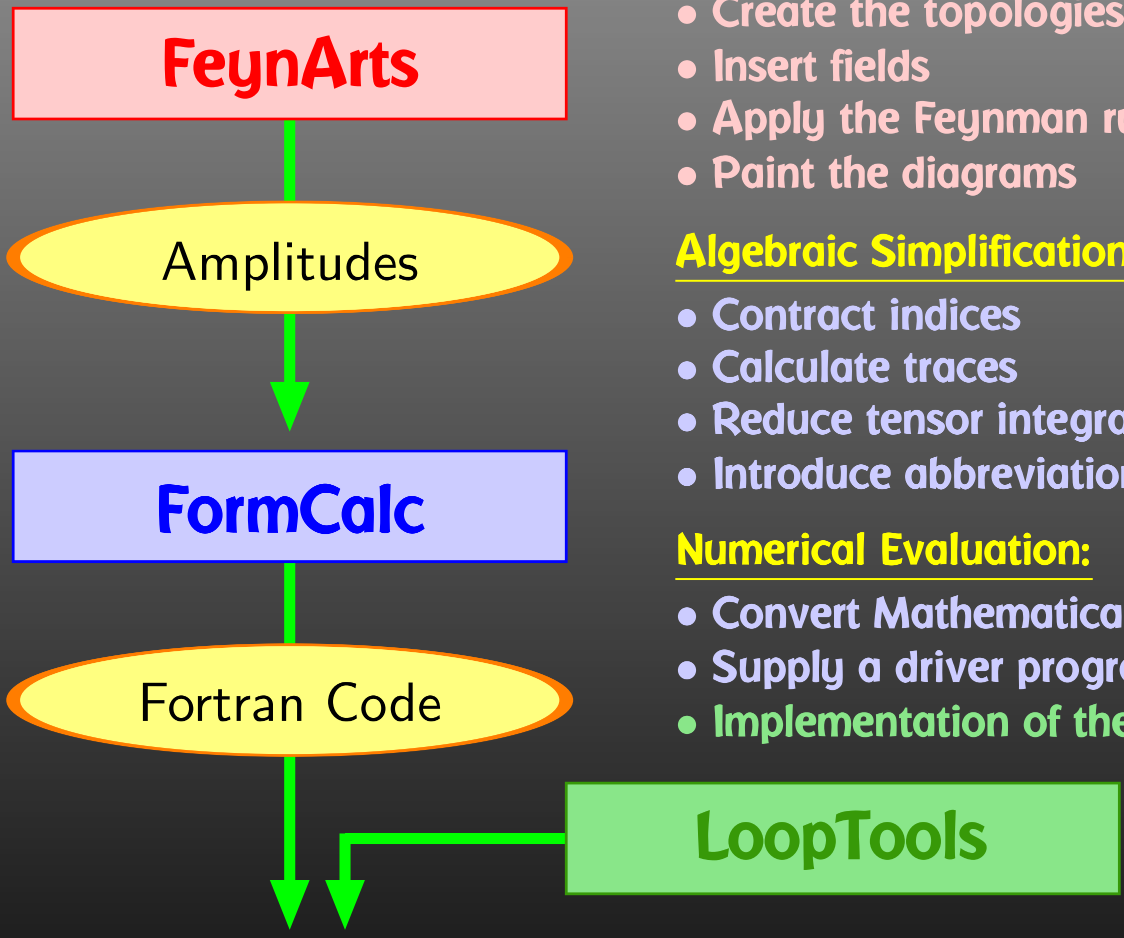


Diagram Generation:

- Create the topologies
- Insert fields
- Apply the Feynman rules
- Paint the diagrams

Algebraic Simplification:

- Contract indices
- Calculate traces
- Reduce tensor integrals
- Introduce abbreviations

Numerical Evaluation:

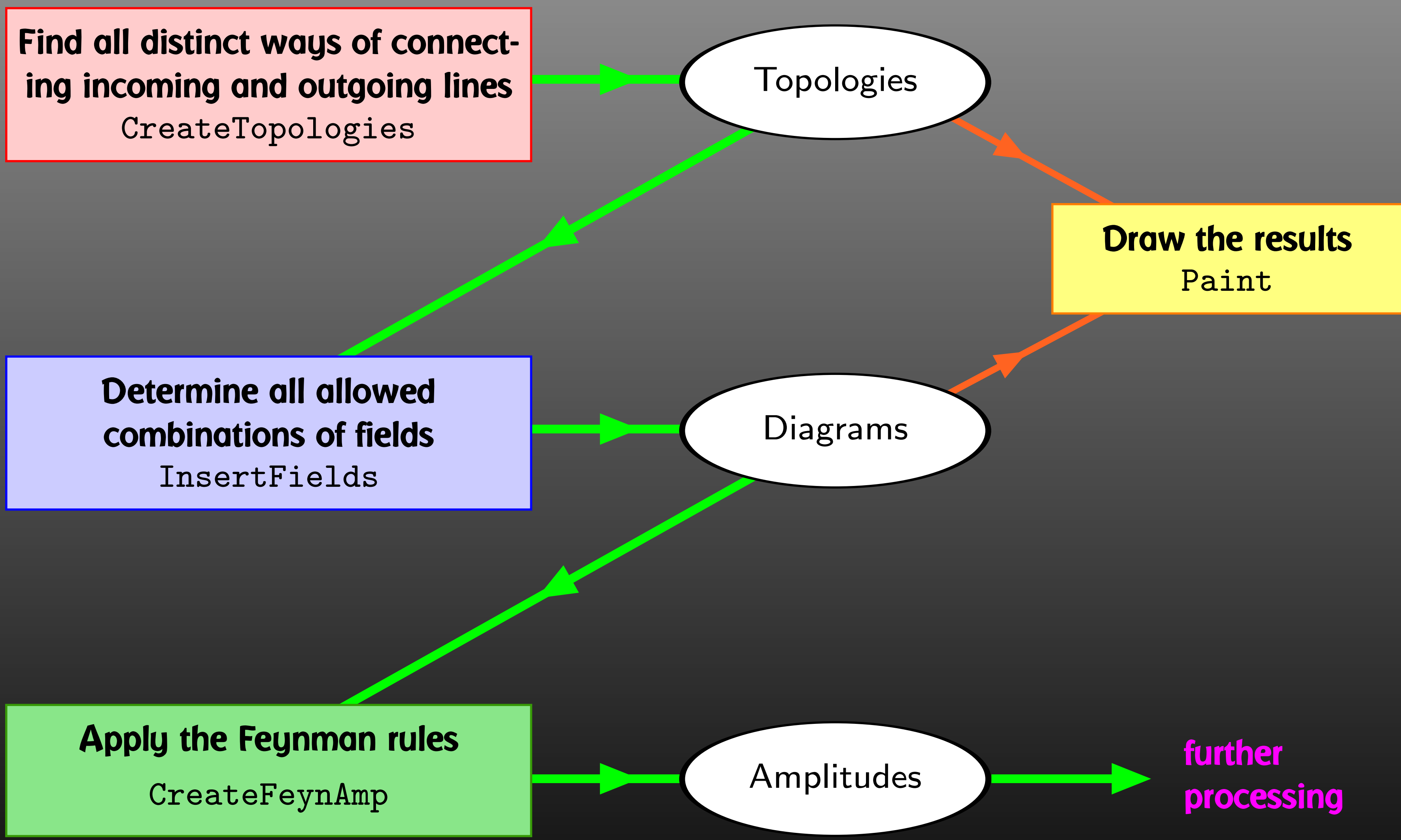
- Convert Mathematica output to Fortran code
- Supply a driver program
- Implementation of the integrals

Symbolic manipulation (Computer Algebra) for the structural and algebraic operations.

Compiled high-level language (Fortran) for the numerical evaluation.

$|\mathcal{M}|^2 \longrightarrow$ **Cross-sections, Decay rates, . . .**

FeynArts



Find all distinct ways of connecting incoming and outgoing lines
CreateTopologies
Topologies
Draw the results
Paint
Determine all allowed combinations of fields
InsertFields
Diagrams
Apply the Feynman rules
CreateFeynAmp
Amplitudes
further processing

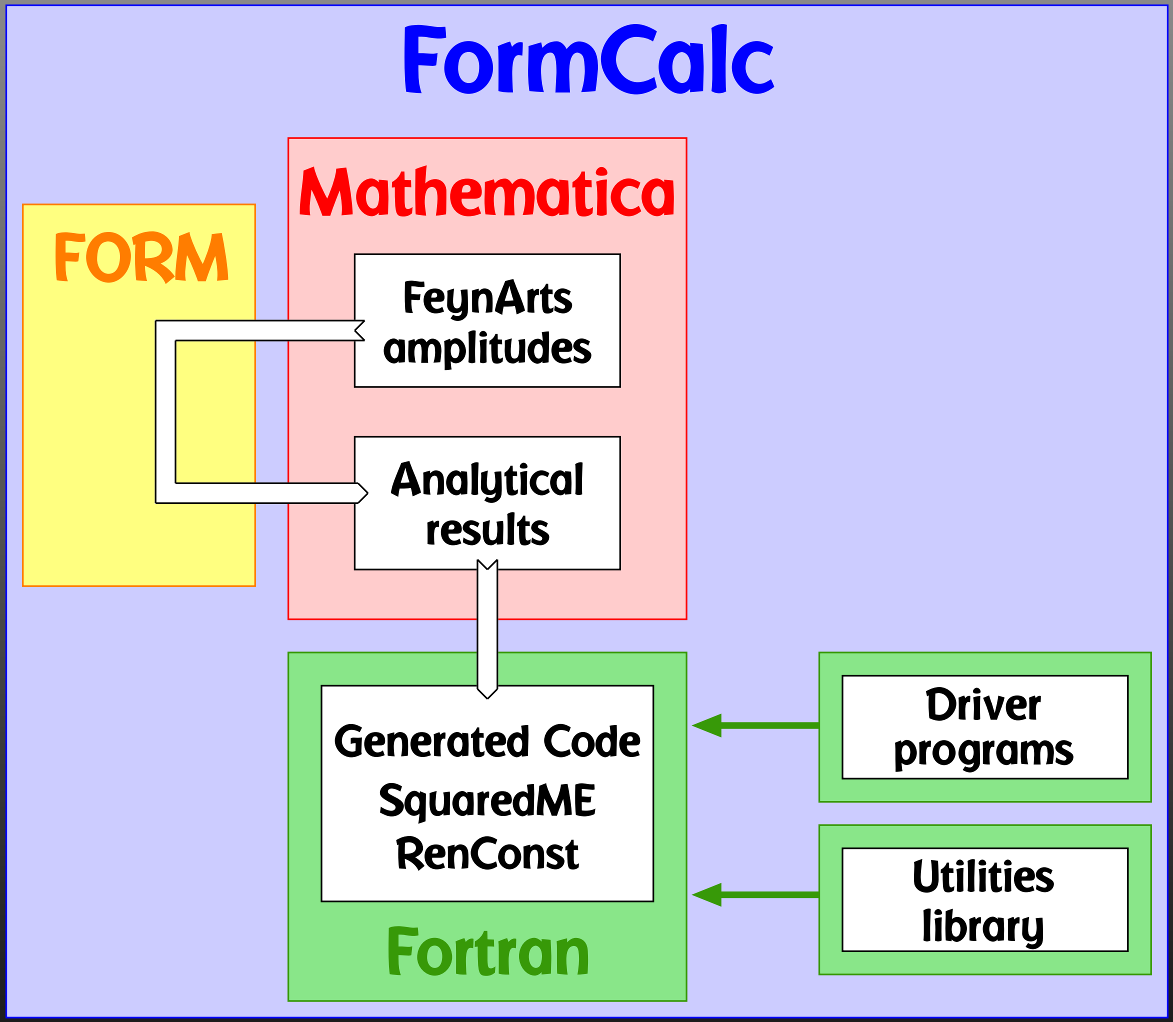
Algebraic Simplification

The amplitudes of `CreateFeynAmp` are in no good shape for direct numerical evaluation.

A number of steps have to be done analytically:

- contract indices as far as possible,
- evaluate fermion traces,
- perform the tensor reduction,
- add local terms arising from D·(divergent integral) (dim reg + dim red),
- simplify open fermion chains,
- simplify and compute the square of SU(N) structures,
- “compactify” the results as much as possible.

FormCalc Internals



FormCalc 8

New Features:

- Significant improvement of algebra through FORM 4 features.
- Vectorization of helicity loop.
- Handling of C code automated.
- OPP optimizations.

Cuba:

- Checkpointing available for all algorithms.

Improvements in the Algebra

- Take advantage of new FORM 4 features: Abbreviationing, Factorization.
- Replace subexpressions by symbols once final (`ToPolynomial`).
- Prevents expansion, preserves (pre-)simplified structure.
- Introduced symbols are largely inert in further operations, thus faster FORM run.
- FormCalc does not use FORM's `format O`n output (yet).
- Factorization (both old/simple and new/full) applied repeatedly.
- Cuts out extra pass to Mathematica as in FormCalc 7.

Improvements in the Algebra

- **Volume of data** returned to Mathematica significantly **smaller** due to 'telescoping effect.'
- Returned (sub)expressions small enough to use **fairly aggressive simplification in Mathematica** within reasonable run-time.
- Several functions can be redefined by the user to **fine-tune simplification:** `FormSub`, `FormDot`, `FormMat`, `FormNum`, `FormQC`.
- **Storage-efficient:** increase reference count rather than insert full copy of subexpression, same as Mathematica's `Share`.
- Generated code **shrinks considerably:** $\mathcal{O}$(30%).

Helicity Loop inherently SIMD

The helicity loop is a fairly obvious candidate for parallel execution, in particular because FormCalc does not insert helicities in the algebra, i.e.

Loop(s) over $\sqrt{s}$ & model parameters

Loop(s) over angular variables

Loop over helicities $\lambda_1, \dots, \lambda_n$

$$\sigma \mathrel{+}= \sum_c C_c\, \mathcal{M}_c^0(\lambda_1, \dots, \lambda_n)^* \, \mathcal{M}_c^1(\lambda_1, \dots, \lambda_n)$$

$$\mathcal{M} = \mathcal{M}(\lambda_1, \lambda_2, \dots) \qquad \text{FormCalc}$$

$$\mathcal{M} = \{\mathcal{M}_{--\dots}, \mathcal{M}_{+-\dots}, \mathcal{M}_{-+\dots}, \mathcal{M}_{++\dots}\} \qquad \text{e.g. GoSam}$$

Helicity sum in FormCalc is thus **SIMD = Single Instruction Multiple Data:** same code $\mathcal{M}$, different data λ_i.

Implementational Issues

Work done in collaboration with J.-N. Lang.

- Overall speedup depends on what fraction of CPU time goes into the helicity loop, thus more efficient for OPP than Pa-Ve (see later).
- `fork/wait` parallelization available since FormCalc 7.5 but competes for compute cores with Cuba. For few cores (e.g. 8), Cuba has higher efficiency.
- Attempted GPU parallelization but not too efficient. Presumably the (fairly expensive) CPU $\to$ GPU transfer of the non-helicity-dependent results outweighs the parallelization gains.
- Vectorization (= several helicity combinations at once) best option on regular x86 hardware with no overhead.

Vectorization in C

- gcc/icc extensions for vector data types only available for real arithmetic, thus have to insert explicit macros for multiplication of complex vectors in C99 (avoid C++ for linking hassles.)
- Code emits explicit SSE3/AVX instructions.
- Max. vector width 1 for SSE3 (2 doubles per operation). Efficient complex multiplication available (2.5 instruct. instead of 6).
- Max. vector width 2 for AVX (i7 Sandy Bridge, 4 doubles per operation).
- Obtained 3.7 out of theoretical speedup 4 with AVX (just the helicity loop).

Vectorization in Fortran

- Uses Fortran 90 vector data types and vector expression syntax.
- Still fixed-format output, can fall back to Fortran 77 through preprocessor defs, e.g. for inclusion in legacy packages.
- Arbitrary choice of vector width because handled by compiler.
- Efficiency depends on compiler optimization, i.e. cannot force particular instruction set.
- No performance figures yet.

Output in C

Up to now: Code generation in Fortran.

Generating C code, available from FormCalc 7, is now mostly automated, i.e. also drivers and utility files are available.

- Switch to C with `SetLanguage["C"]`.
- Only the declarations of the driver code needed to be translated to C, initialization still takes place in Fortran in usual setup (C object files just linked in).
- Private declarations (e.g. for new models) are not automatically translated.
- Setting of compiler flags for CPU type (e.g. AVX on with `-march=corei7-avx`) not yet automated. Default will probably be code generation for host CPU type, to be turned off for generic executable.

OPP Optimizations

Work done in collaboration with E. Mirabella.

We employ the OPP (Ossola, Papadopoulos, Pittau) methods as implemented in the two libraries CutTools and Samurai.

Instead of introducing tensor coefficients, the numerator is put into a subroutine which is sampled by the OPP function, as in:

$$\varepsilon_1^\mu \varepsilon_2^\nu B_{\mu\nu}(p, m_1^2, m_2^2) = B_{\text{cut}}(2, N, p, m_1^2, m_2^2)$$

where $N(q_\mu) = (\varepsilon_1 \cdot q)\,(\varepsilon_2 \cdot q)$.

Compare Pa-Ve: $\varepsilon_1^\mu \varepsilon_2^\nu B_{\mu\nu} = (\varepsilon_1 \cdot \varepsilon_2) B_{00} + (\varepsilon_1 \cdot p)(\varepsilon_2 \cdot p) B_{11}$

OPP Optimizations

Interfacing with CutTools and Samurai quite similar, handled by preprocessor (no re-generation of code necessary).

OPP method generates **fewer terms but nevertheless runs significantly slower** than Passarino-Veltman decomposition.

OPP originally ca. factor 10 slower, now ~ 3.

OPP optimization is work in progress.

OPP Optimizations

One main reason for slowdown: OPP integrals are evaluated **for every** helicity configuration, but **only once** in Pa-Ve.

Observe: **OPP masters** (scalar integrals) depend only on the denominators, so **move them out of helicity loop.**

Currently worked around through LoopTools cache.

More general solution: **Take apart computation of masters and sampling of numerator.** For example:

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

Waiting for Samurai and CutTools folks to adapt API.

OPP Optimizations

- Option to **specify the N** in N-point up to which Passarino–Veltman is used, above OPP.
- **Optimize OPP calls** to reduce sampling effort, e.g. by collecting denominators, as in:

$$\frac{N_4}{D_0 D_1 D_2 D_3} + \frac{N_3}{D_0 D_1 D_2} \to \frac{N_4 + D_3 N_3}{D_0 D_1 D_2 D_3}$$

  Depending on N and rank, this is **not universally better.** Sampling behavior of Samurai and CutTools tabulated and implemented.

- Implementation of link to **Ninja library** in progress, samples fewer times + more stable results.

OPP Optimizations

MadLoop and OpenLoops do this:
Move helicity sum into numerator in interference term,

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

Disadvantages:

- Applicable only if tree-level $\neq 0$.
- Not obvious how to efficiently join with present abbreviation concept.

OPP Optimizations

- Profiler pointed to bottleneck in Fermion Chains. Now evaluated in single inlined function call:

$$\langle u|\, \sigma_\mu \overline{\sigma}_\nu \sigma_\rho \,|v\rangle\, k_1^\mu k_2^\nu k_3^\rho = \langle u|\, k_1 \overline{k}_2 k_3 \,|v\rangle$$

$$\textbf{old} = \mathtt{SxS}(u,\ \mathtt{VxS}(k_1,\ \mathtt{BxS}(k_2,\ \mathtt{VxS}(k_3,\ v))))$$

$$\textbf{new} = \mathtt{ChainV3}(u, k_1, k_2, k_3, v)$$

- Take into account helicity information for massless fermions, as in:

$$\mathtt{Dcut}(3, N, 1 - \mathtt{Hel1}, \dots)$$

Evaluate integrals only if "hel-delta" argument is non-zero.

Cuba Checkpointing

Work done in collaboration with B. Chokoufe.

New version Cuba 3.1 allows checkpointing for all routines.

- Useful for long-running integrations.
- Available only for Vegas so far (3.0).
- Writes complete internal state of integrator to disk in regular intervals.
- Overwrites old state only when new state complete, i.e. crash while writing state recoverable.
- Can recover from last checkpoint, e.g. lose 1 h instead of 1 day after system crash.
- Works regardless of parallelization.

Summary

New Features in FormCalc 8.2: feynarts.de/formcalc

- Better algebra (faster + more compact results) through the use of FORM 4 features.
- Vectorization of helicity loop.
- Output & handling of C code automated.
- OPP optimizations.
- Process specs generated automatically.

Cuba 3.1: feynarts.de/cuba

- Checkpointing for all algorithms.