# Automated one-loop calculations with FormCalc 7

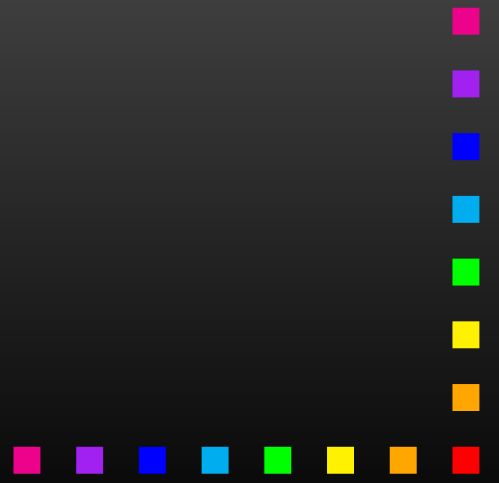
Thomas Hahn

Max-Planck-Institut für Physik
München

From microsoft.com/en-us/windows7:

Why get Version 7?

- To simplify everyday tasks
- To work the way you want
- To do new things



# One-loop since mid-1990s

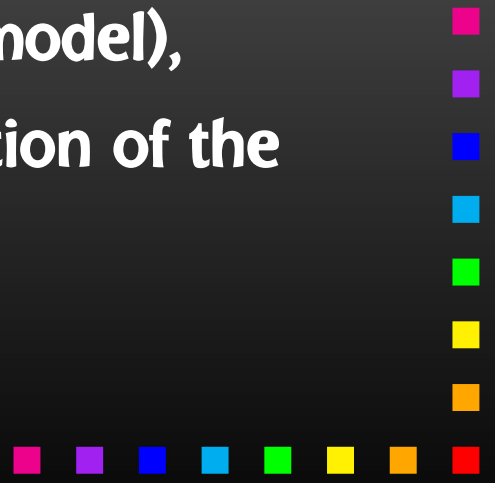
Automated NLO computations is an industry today, with many packages becoming available in the last few years:

- GoSam, HELAC-NLO, aMC@NLO, MadLoop, OpenLoops, BlackHat, Rocket, ...

This report: **FeynArts (1991) + FormCalc (1995)**

FormCalc was doing largely the same as FeynCalc (1992) but used FORM for the time-consuming tasks, hence the name FormCalc.

- Feynman-diagrammatic method,
- Analytic calculation as far as possible (any model),
- Generation of code for the numerical evaluation of the squared matrix element.



# Automated Diagram Evaluation

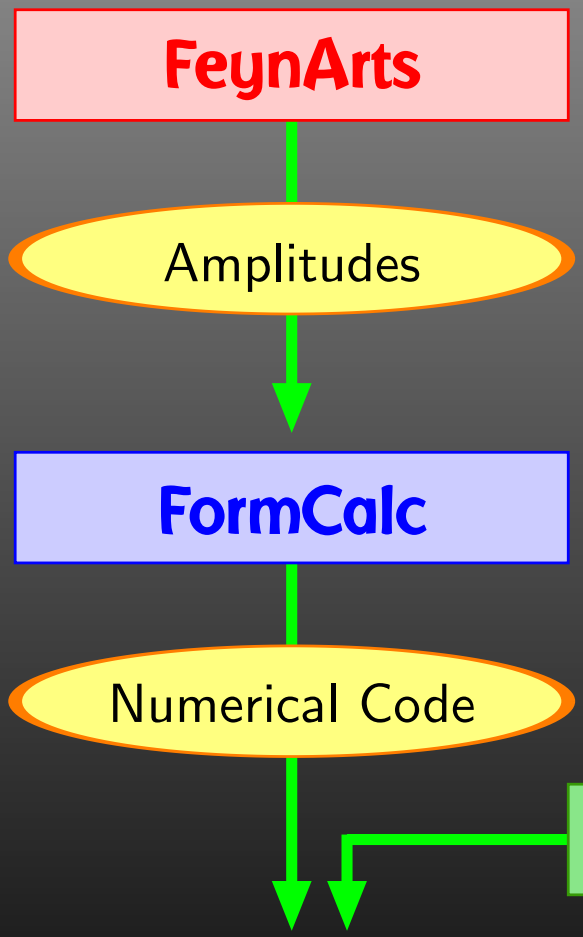



**Diagram Generation:**

- 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:**

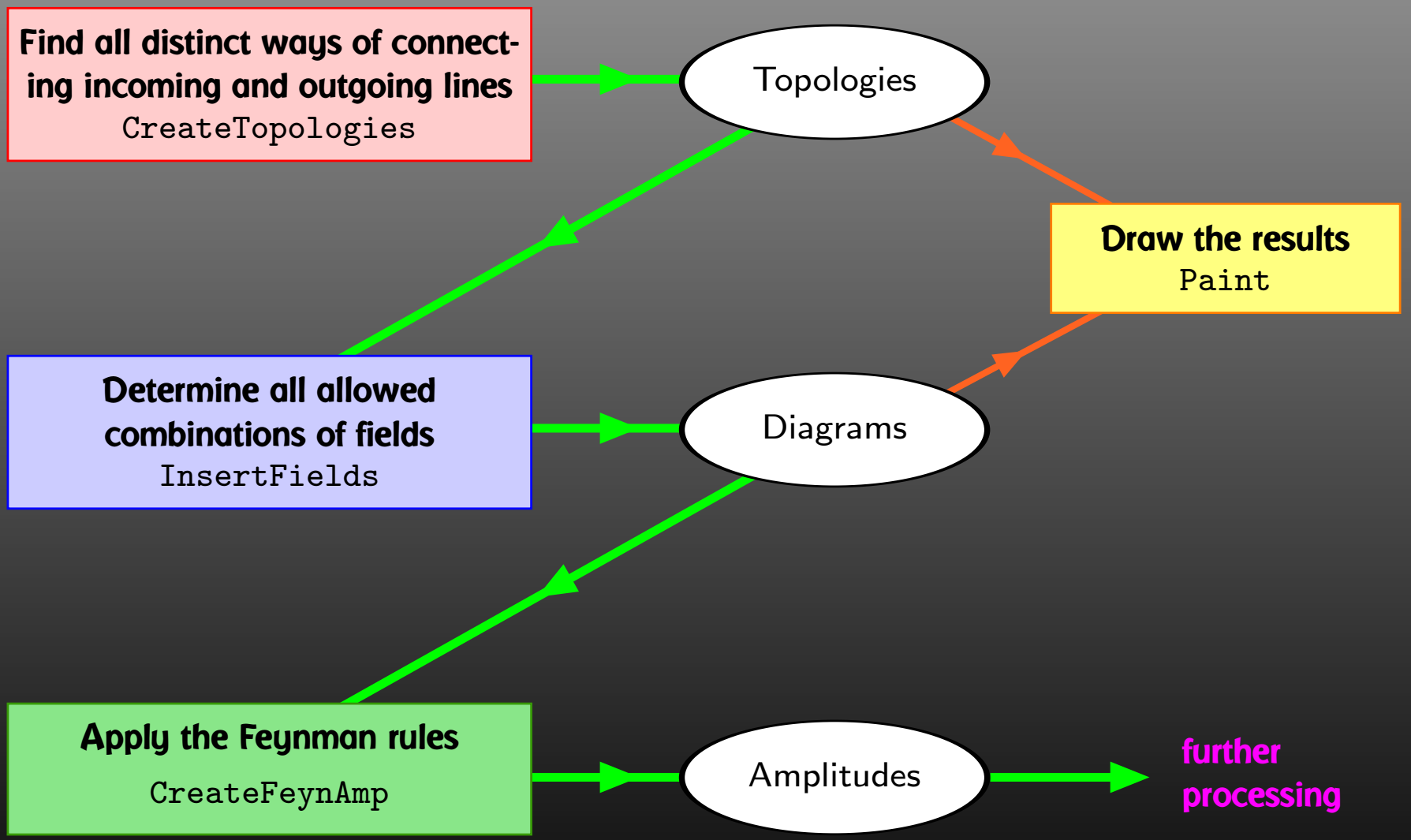
- Convert Mathematica output to numerical code
- Supply a driver program
- Implementation of the integrals

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

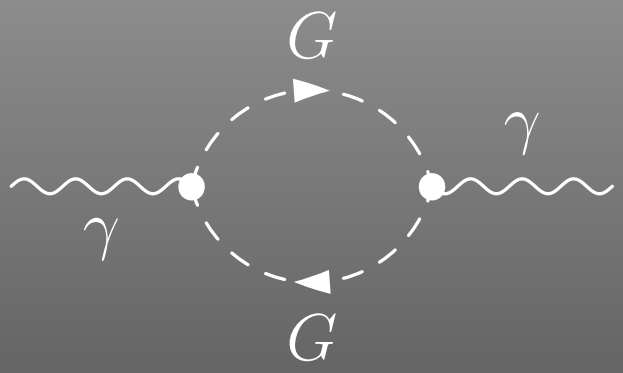
**Compiled high-level language for the numerical evaluation.**

$|\mathcal{M}|^2 \longrightarrow$ **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

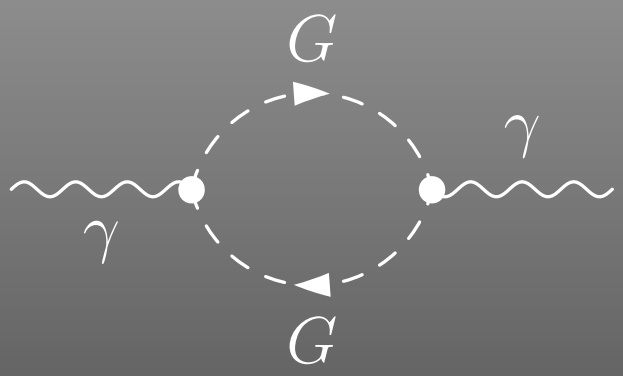
# Sample CreateFeynAmp output




```
= FeynAmp[ identifier ,
           loop momenta ,
           generic amplitude ,
           insertions ]
```

```
GraphID[Topology == 1, Generic == 1]
```

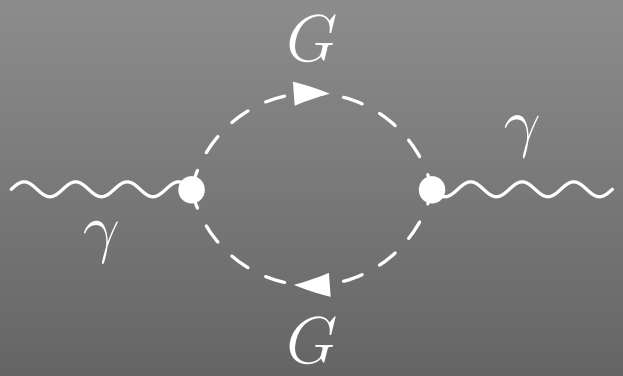
# Sample CreateFeynAmp output




```
= FeynAmp[ identifier,
           loop momenta,
           generic amplitude,
           insertions ]
```

```
Integral[q1]
```

# Sample CreateFeynAmp output




= FeynAmp[ *identifier*,
*loop momenta*,
*generic amplitude*,
*insertions* ]

$\frac{\texttt{I}}{32\ \texttt{Pi}^4}$ RelativeCF ..... *prefactor*

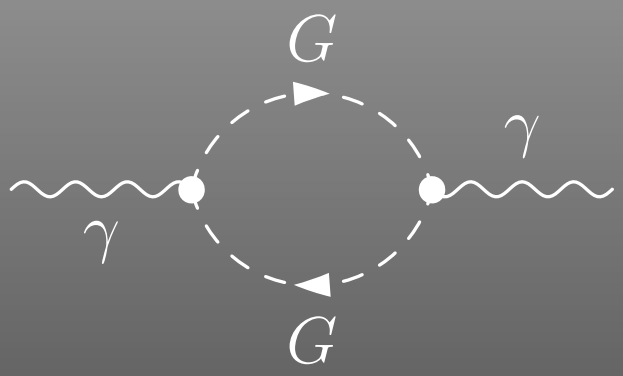
FeynAmpDenominator[$\frac{1}{\texttt{q1}^2 - \texttt{Mass[S[Gen3]]}^2}$, $\frac{1}{(\texttt{-p1 + q1})^2 - \texttt{Mass[S[Gen4]]}^2}$] ..... *loop denominators*

(p1 - 2 q1)[Lor1] (-p1 + 2 q1)[Lor2] ..... *kin. coupling structure*

ep[V[1], p1, Lor1] ep*[V[1], k1, Lor2] ..... *polarization vectors*

$\texttt{G}^{(0)}_{\texttt{SSV}}$[(Mom[1] - Mom[2])[KI1[3]]]
$\texttt{G}^{(0)}_{\texttt{SSV}}$[(Mom[1] - Mom[2])[KI1[3]]] ..... *coupling constants*

# Sample CreateFeynAmp output




= FeynAmp[ *identifier*,
*loop momenta*,
*generic amplitude*,
*insertions* ]

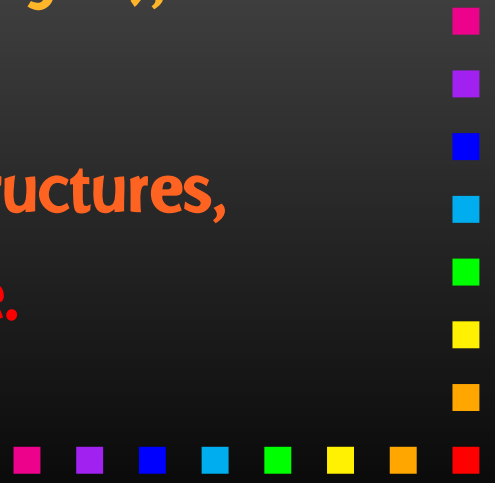
```
{ Mass[S[Gen3]],
  Mass[S[Gen4]],
  G_SSV^(0)[(Mom[1] - Mom[2])[KI1[3]]],
  G_SSV^(0)[(Mom[1] - Mom[2])[KI1[3]]],
  RelativeCF } ->
Insertions[Classes][{MW, MW, I EL, -I EL, 2}]
```

# Algebraic Simplification

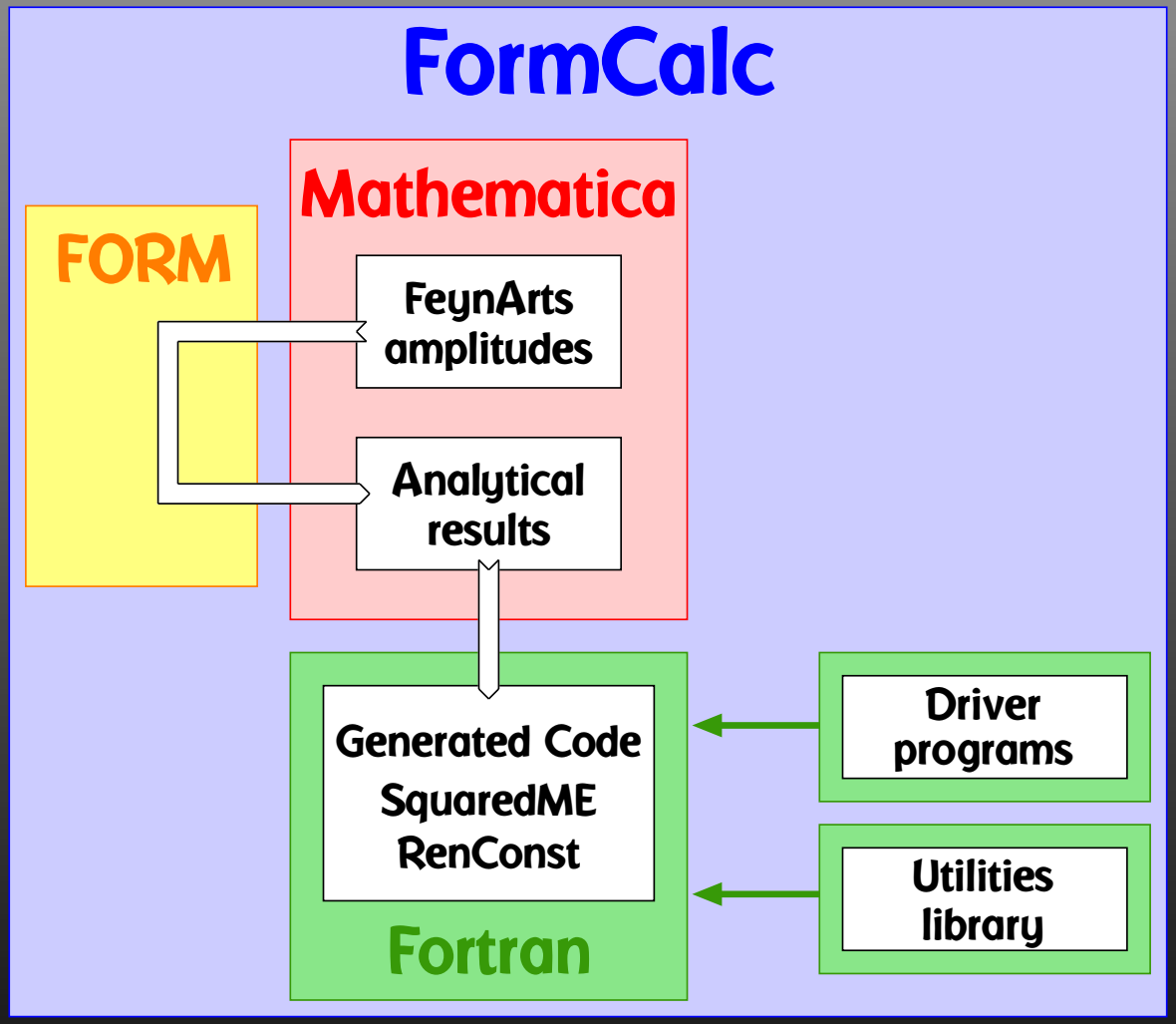
The amplitudes of `CreateFeynAmp` are in no good shape for direct numerical evaluation.

A number of steps have to be done analytically:

- contract indices as far as possible,
- evaluate fermion traces,
- perform the tensor reduction / separate numerators,
- add local terms arising from D·(divergent integral),
- simplify open fermion chains,
- simplify and compute the square of SU(N) structures,
- "compactify" the results as much as possible.



# FormCalc Internals



# FormCalc Output

**A typical term in the output looks like**

```
COi[cc12, MW2, MW2, S, MW2, MZ2, MW2] *
  ( -4 Alfa2 MW2 CW2/SW2 S AbbSum16 +
    32 Alfa2 CW2/SW2 S^2 AbbSum28 +
    4 Alfa2 CW2/SW2 S^2 AbbSum30 -
    8 Alfa2 CW2/SW2 S^2 AbbSum7 +
    Alfa2 CW2/SW2 S(T-U) Abb1 +
    8 Alfa2 CW2/SW2 S(T-U) AbbSum29 )
```

= loop integral

= kinematical variables

= constants

= automatically introduced abbreviations

# Abbreviations

Outright factorization is usually out of question.
Abbreviations are necessary to reduce size of expressions.

```
AbbSum29 = Abb2 + Abb22 + Abb23 + Abb3

Abb22 = Pair1 Pair3 Pair6

Pair3 = Pair[e[3],k[1]]
```

The full expression corresponding to `AbbSum29` is

```
Pair[e[1],e[2]] Pair[e[3],k[1]] Pair[e[4],k[1]] +
Pair[e[1],e[2]] Pair[e[3],k[2]] Pair[e[4],k[1]] +
Pair[e[1],e[2]] Pair[e[3],k[1]] Pair[e[4],k[2]] +
Pair[e[1],e[2]] Pair[e[3],k[2]] Pair[e[4],k[2]]
```

# Categories of Abbreviations

In general, the abbreviations are thus costly in CPU time.

It is key to a decent performance that the abbreviations are separated into different Categories:

- Abbreviations that depend on the helicities,
- Abbreviations that depend on angular variables,
- Abbreviations that depend only on $\sqrt{s}$.

Correct execution of the categories guarantees that almost no redundant evaluations are made and makes the generated code essentially as fast as hand-tuned code.

# Fermion Chains

FormCalc uses Dirac (4-component) spinors in most of the algebra (extension to $D$ dim more obvious).

Move to 2-comp. Weyl spinors for the numerical evaluation:

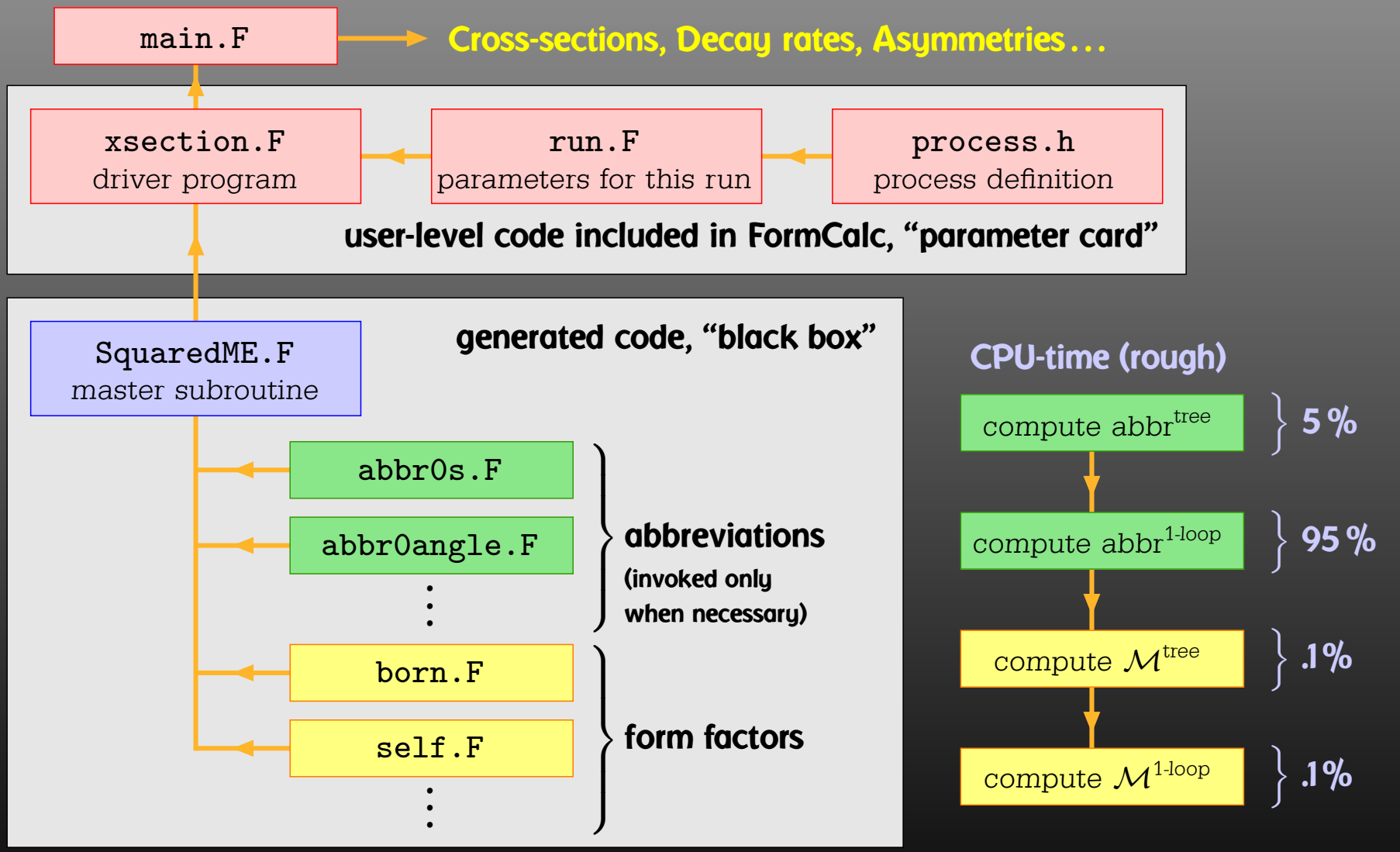
$$\langle u|_4 \equiv \left(\langle u_+|_2\,, \langle u_-|_2\right), \qquad |v\rangle_4 \equiv \begin{pmatrix} |v_-\rangle_2 \\ |v_+\rangle_2 \end{pmatrix}.$$

Every chiral Dirac chain maps onto a ***single*** Weyl chain:

$$\langle u|\,\omega_-\gamma_\mu\gamma_\nu\cdots|v\rangle = \langle u_-|\,\overline{\sigma}_\mu\sigma_\nu\cdots|v_\pm\rangle\,,$$
$$\langle u|\,\omega_+\gamma_\mu\gamma_\nu\cdots|v\rangle = \langle u_+|\,\sigma_\mu\overline{\sigma}_\nu\cdots|v_\mp\rangle\,.$$

FORM-like notation: $\langle u|\,\sigma_\mu\overline{\sigma}_\nu\sigma_\rho\,|v\rangle\,k_1^\mu\varepsilon_2^\nu k_3^\rho \equiv \langle u|\,k_1\overline{\varepsilon}_2 k_3\,|v\rangle\,.$

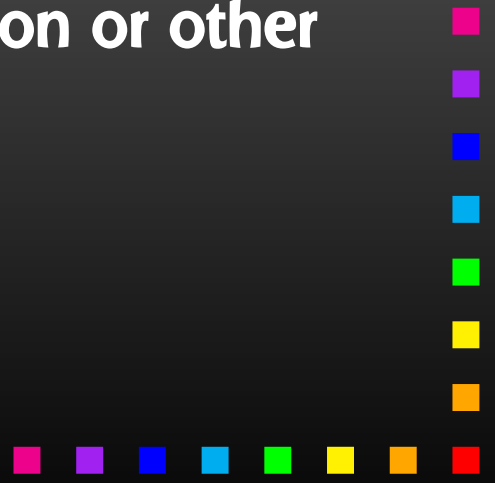
# Numerical Evaluation



# The Case for Mathematica

Several packages make it a selling point that they are "100% free of Mathematica."

- Use of Mathematica is a feature, not a bug.
- Mathematica's language makes it easy for the user to examine and modify results, without having to contact the package authors.
- Mathematica is known and available to most physicists.
- Mathematica is far more powerful than Python or other free symbolic languages.



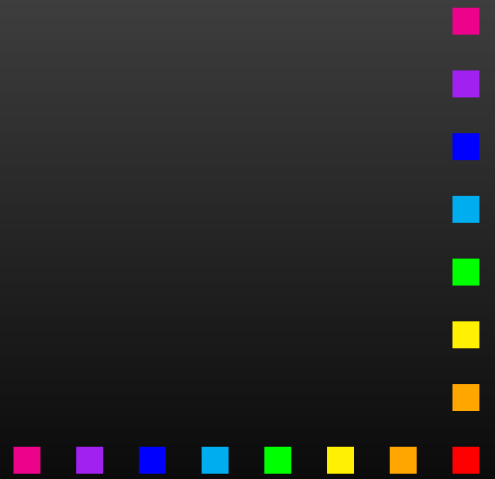
# FormCalc 7 – 'The Parallel Release'

New Features:

- Unitarity methods (OPP),
- Parallelization of the helicity loop,
- C output and Improved code generation,
- Command-line parameters for model initialization, MSSM (SM) initialization via FeynHiggs,
- Analytic tensor reduction,
- Auxiliary functions for operator matching.

Cuba:

- Built-in Parallelization.

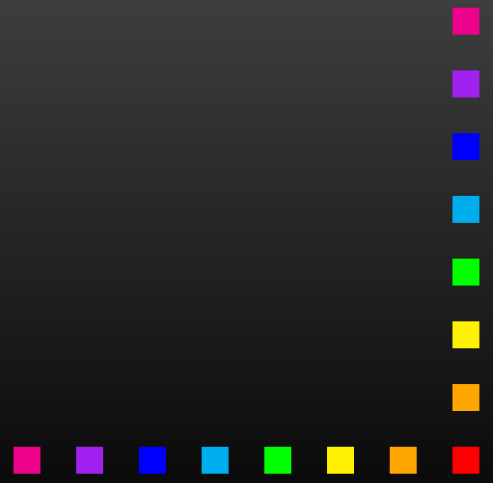


# Unitarity Debunk

Many packages claiming to use Unitarity Methods in fact perform perfectly ordinary Feynman-diagrammatic computations with the usual double-factorial growth.

They use Unitarity Methods 'only' for the computation of the tensor integrals, i.e. instead of Passarino-Veltman tensor decomposition.

FormCalc is no different.



# Unitarity Methods

*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

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

# Unitarity Methods

So far tested on a handful of $2 \to 2$ and $2 \to 3$ processes, get agreement to about 10 digits.

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

Performance somewhat wanting as of now, Passarino-Veltman beats naive OPP hands-down in the processes we looked at.

Main problem: OPP integrals are evaluated for every helicity configuration, but only once in Passarino-Veltman decomposition.

OPP optimization is work in progress.

# Optimizing OPP Performance

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

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

- Switch off simplifications that break up loop integrals, e.g.

$$\frac{q^2}{(q^2 - m^2) D_1 D_2} = \frac{1}{D_1 D_2} + \frac{m^2}{(q^2 - m^2) D_1 D_2}$$

Better performance calling OPP routine once with more complicated integral than twice with simpler integrals.

# Optimizing OPP Performance

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

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

Disadvantages:

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

# Optimizing OPP Performance

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

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

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

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

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

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

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

# Parallelization of Helicity Loop

Perhaps the most obvious way to tackle the OPP-induced slowdown is to parallelize the helicity loop.

FormCalc does not insert helicities in the algebra, i.e.

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

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

Evaluation of matrix element in FormCalc is thus classical **SIMD: Single Instruction Multiple Data.**

Run same code ($\mathcal{M}$) for different data ($\lambda_i$).

At least conceptually simple to parallelize.

# Implementational Issues

- Organize variables such that only helicity-dependent ones need to be transferred to workers. Changes in LoopTools necessary to control cached loop integrals.
- Reorganization of squared matrix element computation and actual parallelization code fairly straightforward.
- Uses same fork-based method as Cuba, details see later.
- Not clear yet how best to divide cores between this and Cuba's parallelization, at least on a single CPU. Perhaps go to GPU.
- Brand-new implementation, no performance figures yet, preliminary results promising.

# Code generation

Traditionally: Output in Fortran.
Code generator is rather sophisticated by now, e.g.

- **Expressions too large** for Fortran are split into parts, as in

```
var = part1
var = var + part2
...
```

- **High level of optimization,** e.g. common subexpressions are pulled out and computed in temporary variables.
- **Many ancillary functions** make code generation versatile and highly automatable, such that the resulting code needs few or no changes by hand.

Example: a significant part of FeynHiggs has been generated this way.

# C Output and Improvements in Code Generation

- **Output in C99** available now, makes integration into C/C++ codes easier and allows for GPU programming.

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

- Code better structured, e.g.
  - **Loops and tests handled through macros,** e.g.
    `LOOP(var, 1, 10, 1) ... ENDLOOP(var)`
  - **Sectioning by comments,** to aid **automated substitution** e.g. with `sed`, e.g.
    `* BEGIN VARDECL ... * END VARDECL`
  - Introduced **data types** `RealType` **and** `ComplexType` for better abstraction, can e.g. be changed to different precision.

# Command-line parameters for model initialization

Extension of command-line argument parsing:

```
run :arg1 :arg2 ... uuuuu 0,1000
```

The ':'-arguments are passed to model initialization code.

Internal routines in `xsection.F` accordingly have additional parameters `argv`, `argc`.

Application: FeynHiggs as Frontend for FormCalc-generated code (`model_fh.F`)

```
run :fhparameterfile :fhflags uuuuu 0,1000
```

- FeynHiggs initializes MSSM (SM) parameters and passes them to FormCalc code.
- No duplication of initialization code.
- Parameters consistent between Higgs-mass and cross-section computation.

# Analytic Tensor Reduction

*Work done in collaboration with S. Agrawal.*

Passarino-Veltman reduction is still useful. So far:

- introduction of tensor coefficients in FormCalc, e.g.

$$\int \mathrm{d}^4 q \frac{q_\mu q_\nu}{D_0 D_1} \sim B_{\mu\nu} = g_{\mu\nu} B_{00} + p_\mu p_\nu B_{11}$$

- complete reduction to scalars only numerically in LoopTools.

**Available now: Analytic Reduction in FormCalc.**

```
CalcFeynAmp[..., PaVeReduce -> True]
```

# Analytic Tensor Reduction

Reduction formulas from Denner & Dittmaier, hep-ph/0509141. Not straightforward to implement in FORM.

Apart from analytic considerations, this is useful e.g. for low-energy observables, where small momentum transfer may lead to numerical instabilities in numerical reduction, as in:

$$B_\mu = p_\mu B_1 \quad \text{for} \quad p \to 0$$

Unless FormCalc finds a way to cancel it immediately, the inverse Gram determinant appears wrapped in `IGram` in the output, so is available for further modifications.

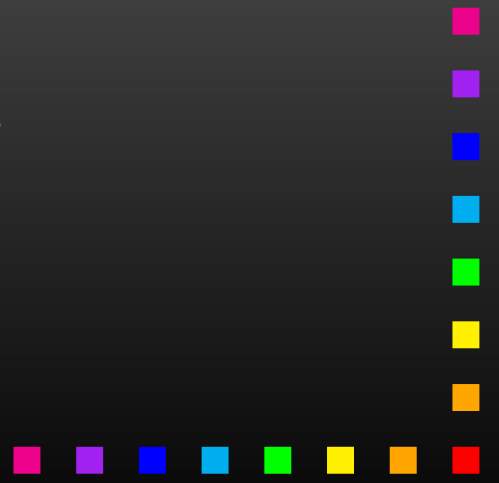
# Aiding Operator Matching

As numerical calculations are done mostly using Weyl-spinor chains, there has been a paradigm shift for Dirac chains to make them better suited for analytical purposes, e.g. the extraction of Wilson coefficients.

- The `FermionOrder` option of `CalcFeynAmp` implements Fierz methods for Dirac chains, allowing the user to force fermion chains into any desired order. This includes the `Colour` method which brings the spinors into the same order as the external colour indices.
- The `Antisymmetrize` option allows the choice of completely antisymmetrized Dirac chains, i.e. `DiracChain`$[-1, \mu, \nu] = \sigma_{\mu\nu}$.
- The `Evanescent` option tracks operators before and after Fierzing for better control of $\varepsilon$-dimensional terms.

# Cuba Parallelization: Design Considerations

- 1 Master, $N$ workers on $N$-core system.
  Master generates all samples, thus no issues with seeding random-number generators.
- No parallelization across the network (e.g. via MPI).
  OS functions only, no extra software needed.
  Mathematica separate: re-define `MapSample` e.g. by `ParallelMap`.
- Uses internal cores 'only', thus e.g. 4 or 8.
  (Many) more cores not necessarily useful since speed-ups not expected to be linear.
- Auto-detect # of cores + load at run-time.
  User control through environment variable `CUBACORES`.
  No re-compile necessary.



# fork vs. pthread_create

- `pthread_create` creates additional thread in same memory space.
- `fork` creates completely independent process.
- Must use `fork` for non-reentrant integrands.
  Reentrancy cannot be fully controlled e.g. in Fortran.
- Keep `fork` calls minimal: 'Spinning Threads' method
  = fork $N$ times at entry into Cuba routine.

  No `fork` in Windows, Cygwin emulates but quite slow.
  Despite 'copy-on-write' (Linux), `fork` is moderately 'expensive' even on Linux/MacOS.
- Master-worker communication:
  (if available:) shared memory for samples,
  socketpair I/O for control information (creates scheduling hint for kernel, too).

# Implementation

- **Main sampling routine** `DoSample` **already abstracted in Cuba 1, 2 since C/C++ and Mathematica implementations very different.**
- `DoSample` **straightforward to parallelize on** $N$ **cores:**

  **Serial** $\rightarrow$ **sample** $n$ **points**

  **Parallel** $\rightarrow$ **send** $\lceil n/N \rceil$ **points to core 1**
  $\rightarrow$ **send** $\lceil n/N \rceil$ **points to core 2**
  $\rightarrow \ldots$

- **Fill fewer cores if not enough samples.**
- **Divonne: Parallelizing** `DoSample` **alone not satisfactory. Speed-ups generally** $\lesssim$ **1.5.**
  **Partitioning phase significant. Originally recursive, had to 'un-recurse' algorithm first.**

# Inefficiencies

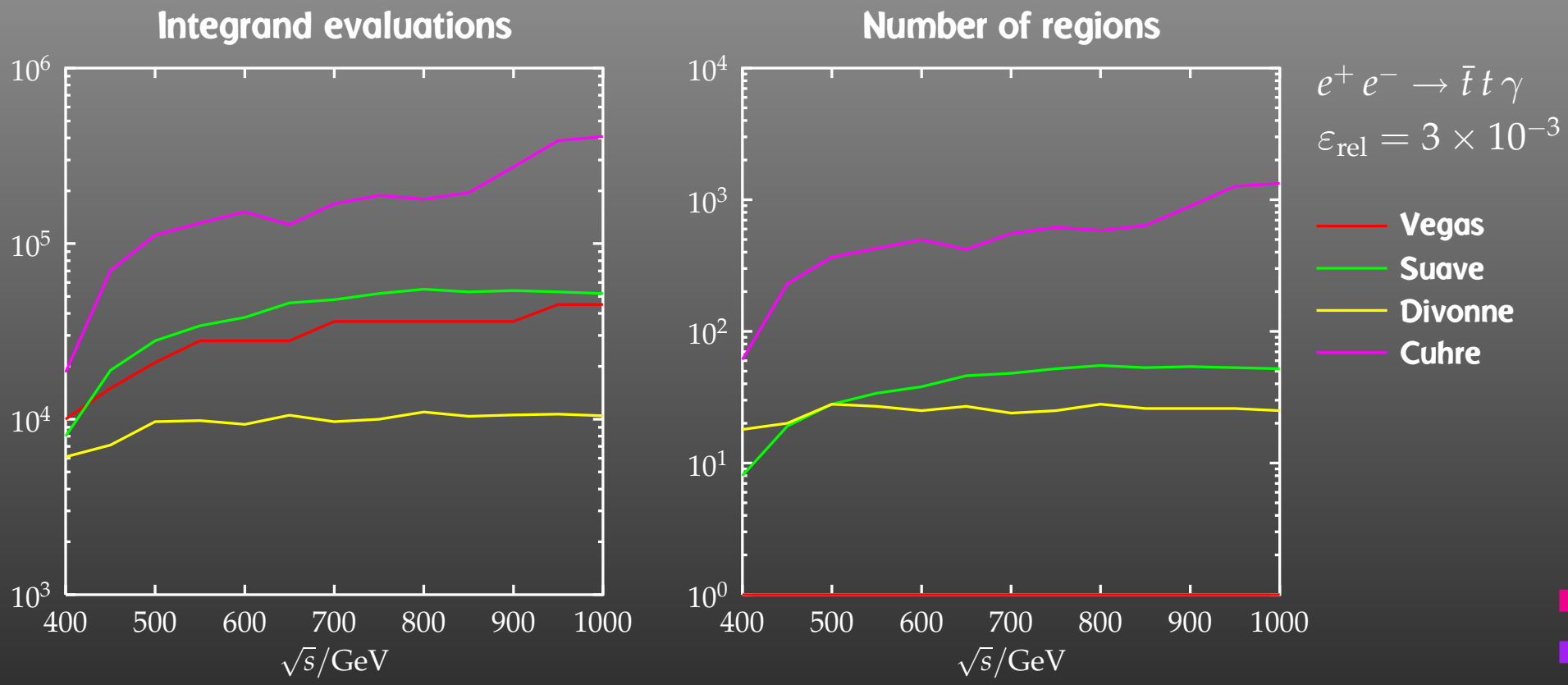
Assess parallelization efficiency through

$$\text{speed-up} = \frac{t_{\text{serial}}}{t_{N\text{-cores}}} \quad \text{ideally} = N.$$

- **Parallelization overhead** = Extra time for communication, scheduling efficiency etc.
  Overhead can be estimated through $t_{\text{serial}}/t_{\text{1-core}} < 1$.
- **Load levelling** = Keeping cores busy. If only $N - n$ busy, absolute timing may be ok but $N$-core speed-up lousy.
- Caveat: Hyperthreading, e.g. i7 has 8 virtual, 4 real cores.

Speed-ups will obviously depend on the 'cost' of the integrand: The more time a single integrand evaluation takes, the better speed-ups can be expected to achieve.

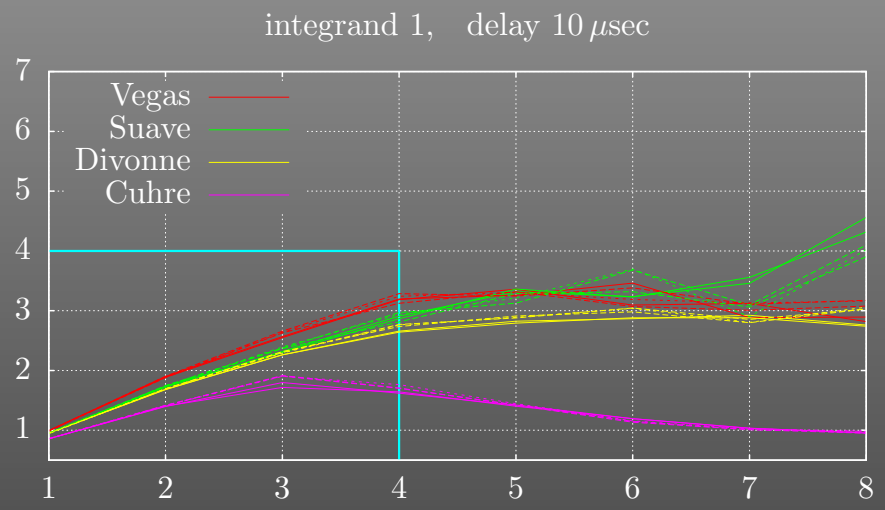
# Cuba Comparison

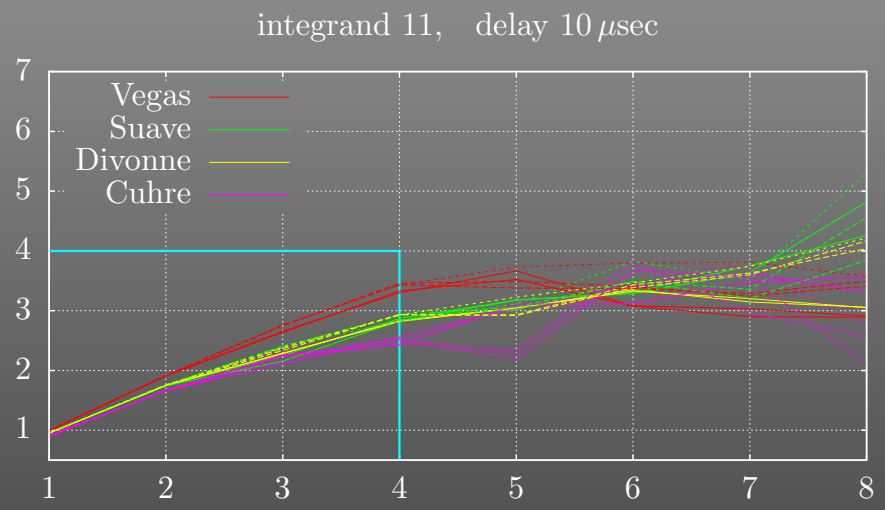


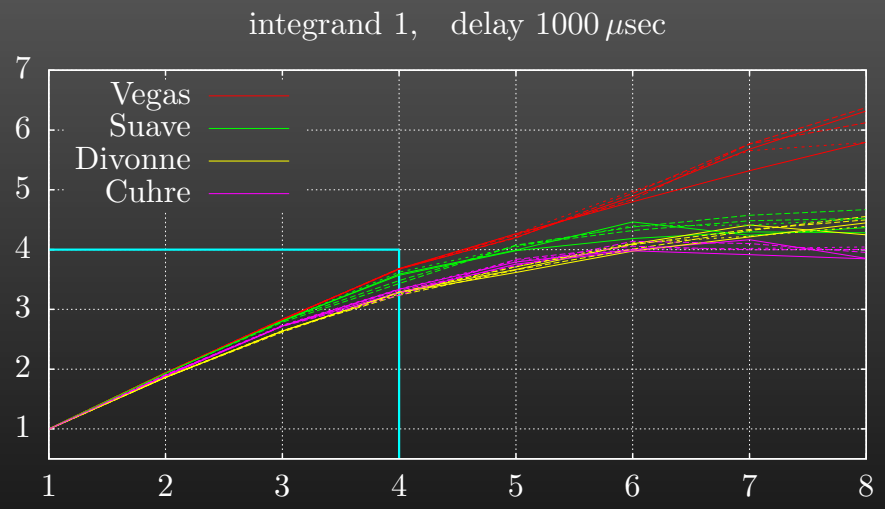

'Gauge' integration problem first:

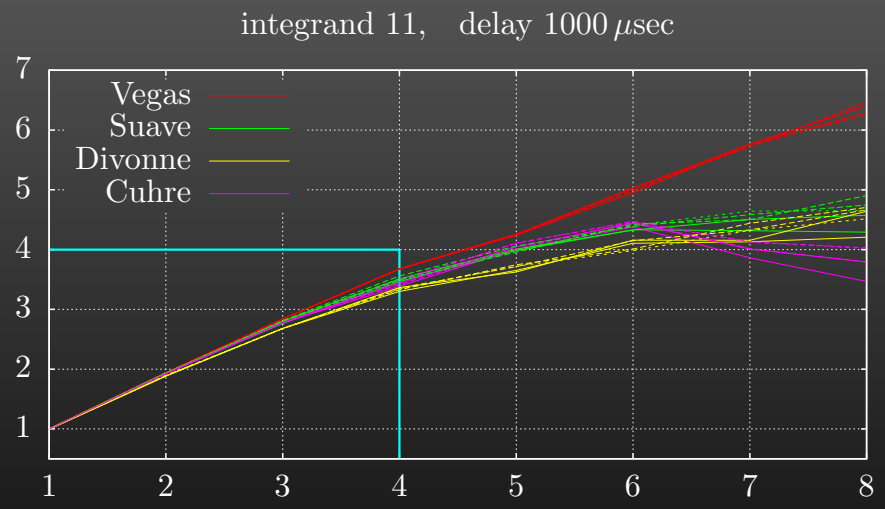
- Compute with all four routines.
- Check whether results are consistent.
- Select fastest algorithm.

# Timing Results




$$f_1 = \sin x \, \cos y \, \exp z$$

$$\varepsilon_{\text{rel}} = 10^{-4}$$

$$f_{11} = \Theta(1 - x^2 - y^2 - z^2)$$

# Summary

New Features in FormCalc 7: feynarts.de/formcalc

- Unitarity methods (OPP) using Samurai or CutTools,
- Parallelized helicity loop,
- C output and Improved code generation,
- Command-line parameters for model initialization,
- Initialization of MSSM parameters via FeynHiggs,
- Analytic tensor reduction in `CalcFeynAmp`,
- Options aiding operator matching (Fierz, antisymmetry, evanescent operators).

Cuba: feynarts.de/cuba

- Built-in Parallelization available simply by compiling with Cuba 3.