FormCalc 7

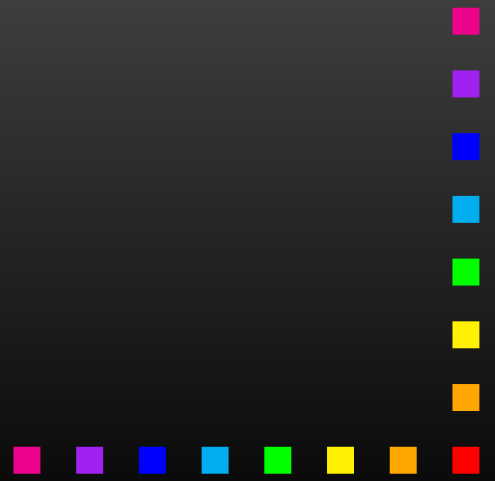
Thomas Hahn

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



Automated Diagram Evaluation

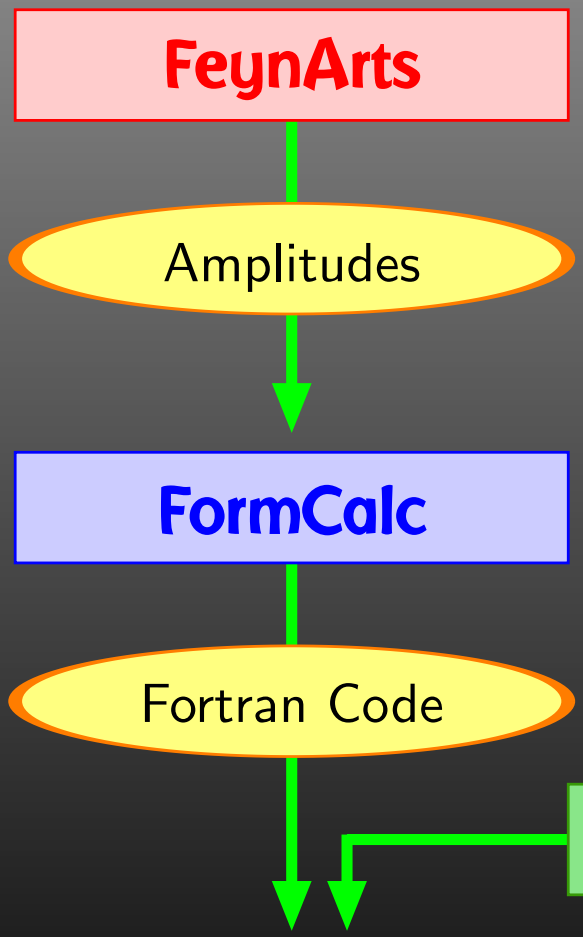


Diagram Generation:

- 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:

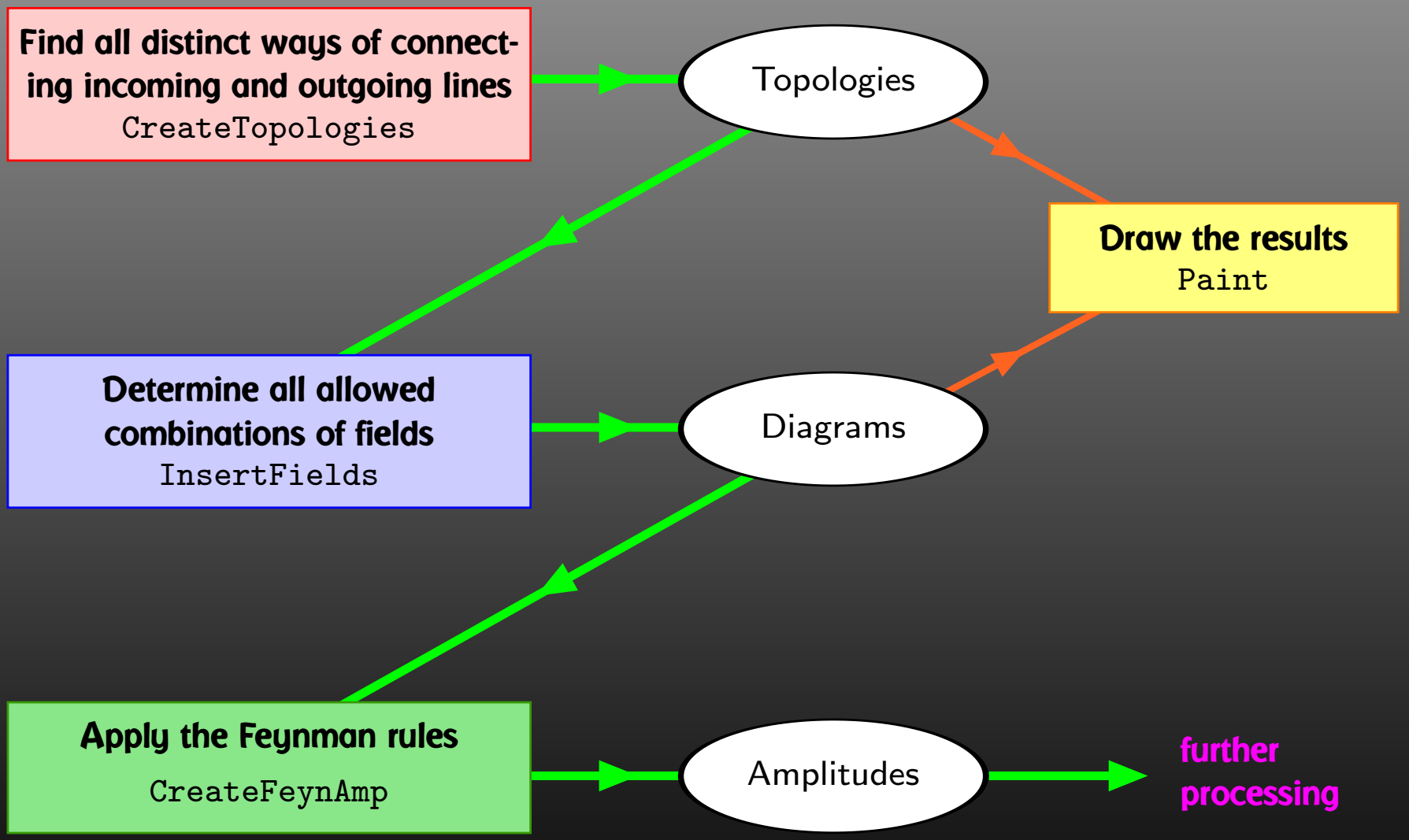
- Convert Mathematica output to Fortran code
- Supply a driver program
- Implementation of the integrals

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

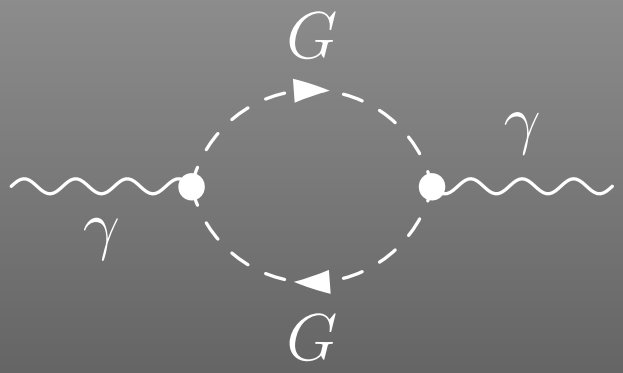
Compiled high-level language (Fortran) for the numerical evaluation.

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

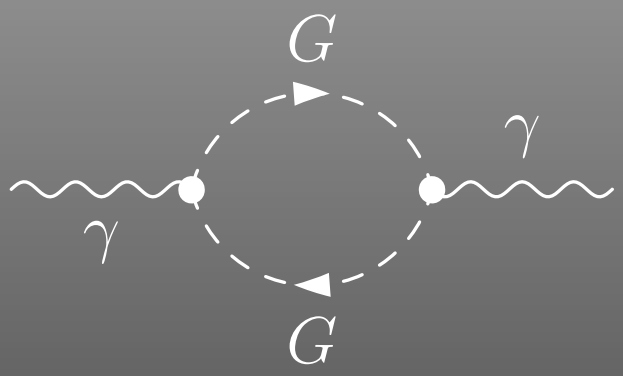
Sample CreateFeynAmp output




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

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

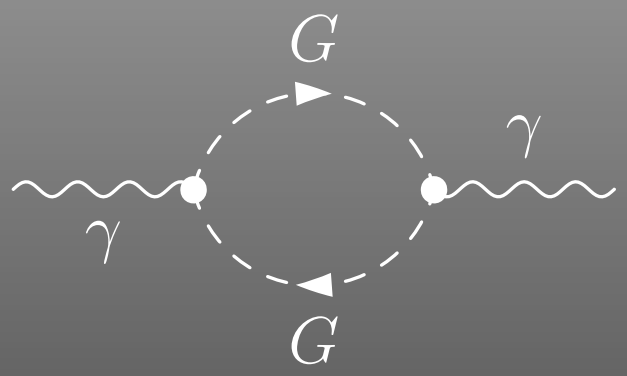
Sample CreateFeynAmp output




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

```
Integral[q1]
```

Sample CreateFeynAmp output



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

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

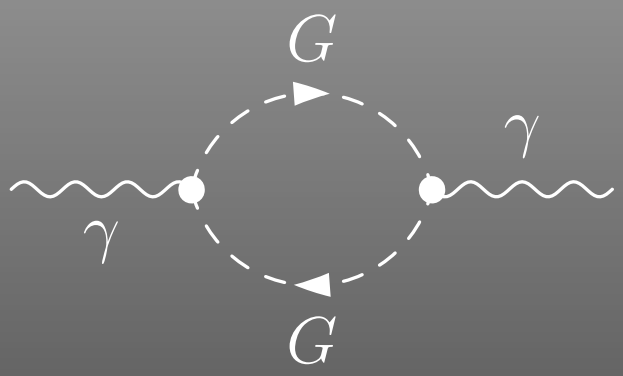
FeynAmpDenominator[$\frac{1}{\texttt{q1}^2 - \texttt{Mass[S[Gen3]]}^2}$,
$\frac{1}{(\texttt{-p1 + q1})^2 - \texttt{Mass[S[Gen4]]}^2}$] ..... *loop denominators*

(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}]
```

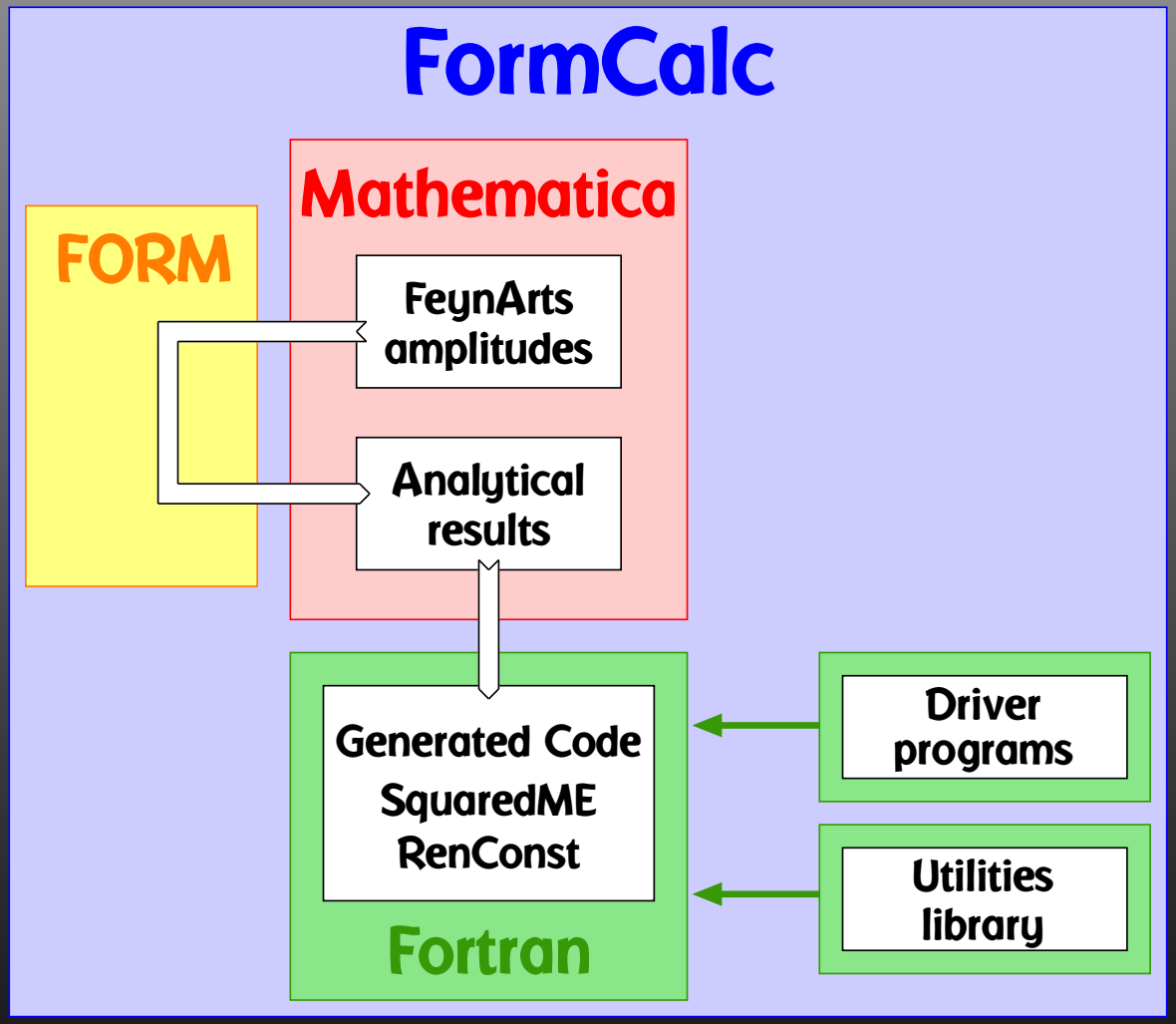
Algebraic Simplification

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,
- 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 Output

A typical term in the output looks like

```
C0i[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]]
```

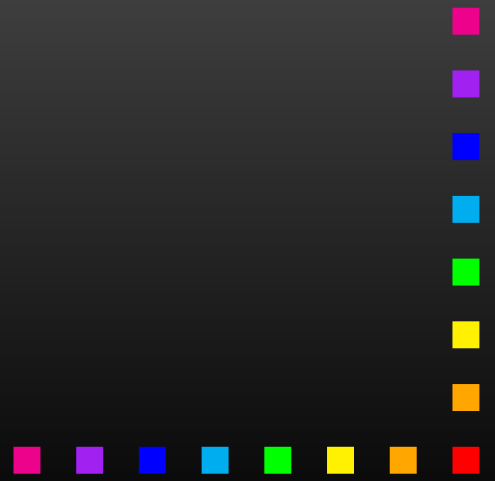
FormCalc 7

New Features:

- Analytic tensor reduction,
- Unitarity methods (OPP),
- Improved code generation,
- Command-line parameters for model initialization, MSSM (SM) initialization via FeynHiggs.
- Auxiliary functions for operator matching.

Cuba:

- Built-in Parallelization.



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.

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

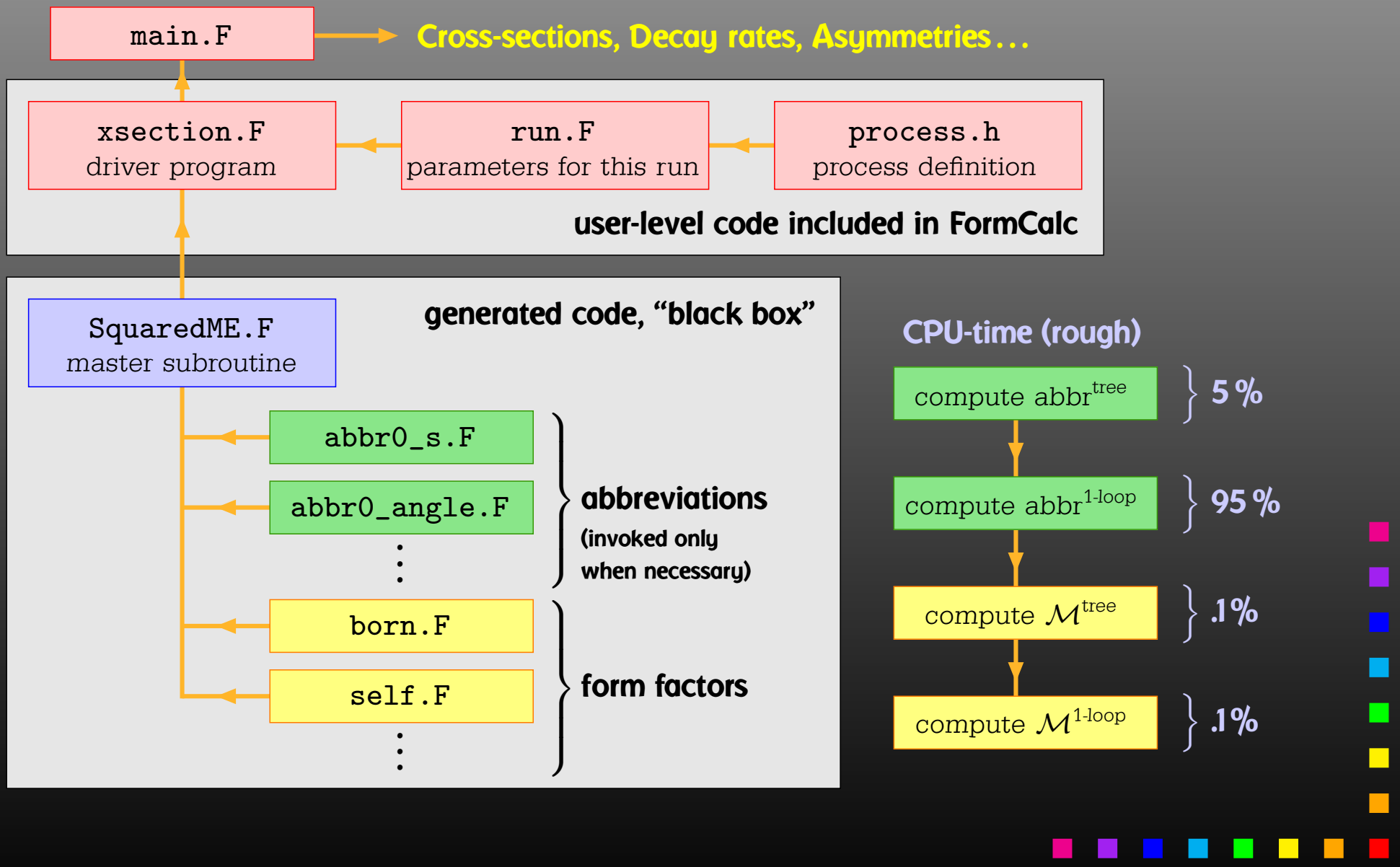
So far tested on a handful of $2 \to 2$ and $2 \to 3$ processes, get agreement to about 10 digits.

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

Currently optimizing performance:

- Option to specify the N in N-point up to which Passarino-Veltman is used, above OPP.
- Minimizing OPP calls to reduce sampling effort - work in progress.
- Already looked into tweaking caching of loop integrals, but pointless: lower-N integrals also needed by OPP.

Numerical Evaluation in Fortran 77



Code generation

Currently: Output in Fortran 77.
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.

Improvements in Code Generation

- **Output in C** largely finished, makes integration into C/C++ codes easier and allows for GPU programming.
- **Loops and tests handled through macros,** e.g.

```
LOOP(var, 1,10,1)
ENDLOOP(var)
```

- Main subroutine `SquaredME` now **sectioned 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.

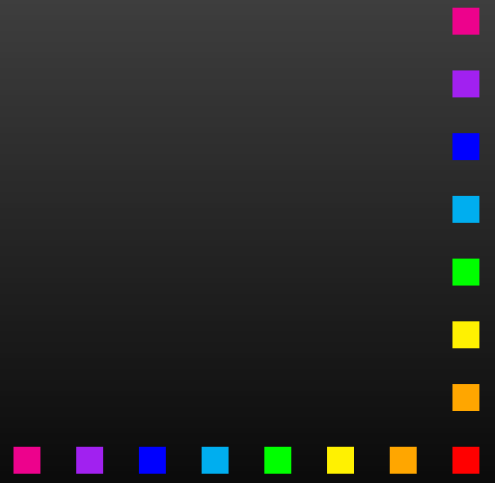
Command-line parameters for model initialization

Extension of command-line argument parsing:

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

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

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

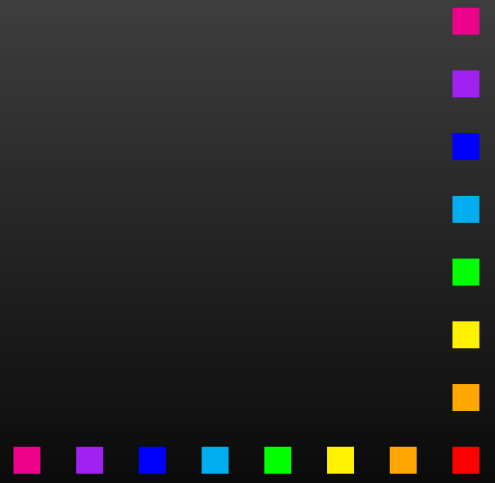


Model Initialization through FeynHiggs

- `model_fh.F` **uses FeynHiggs as Frontend for FormCalc-generated code:**

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

- **FeynHiggs initializes MSSM (SM) parameters and passes them to FormCalc code.**
- **No duplication of initialization code.**
- **Parameters consistent between Higgs-mass computation and cross-section calculation.**
- **Needs FeynHiggs 2.8.1 or above.**



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 ε-dimensional terms.

Not the Cross-Section

Example: extract the Wilson coefficients for $b \to s\gamma$.

```
tops = CreateTopologies[1, 1 -> 2]
ins = InsertFields[tops, F[4,{3}] -> {F[4,{2}], V[1]}]
vert = CalcFeynAmp[CreateFeynAmp[ins], FermionChains -> Chiral]

mat[p_Plus] := mat/@ p

mat[r_. DiracChain[s2_Spinor, om_, mu_, s1:Spinor[p1_, m1_, _]]] :=
  I/(2 m1) mat[r DiracChain[sigmunu[om]]] +
  2/m1 r Pair[mu, p1] DiracChain[s2, om, s1]

mat[r_. DiracChain[sigmunu[om_]], SUNT[Col1, Col2]] :=
  r O7[om]/(EL MB/(16 Pi^2))

mat[r_. DiracChain[sigmunu[om_]], SUNT[Glu1, Col2, Col1]] :=
  r O8[om]/(GS MB/(16 Pi^2))

coeff = Plus@@ vert //. abbr /. Mat -> mat

c7 = Coefficient[coeff, O7[6]]
c8 = Coefficient[coeff, O8[6]]
```

Not the Cross-Section

Using FormCalc's output functions it is also pretty straightforward to generate your own Fortran code:

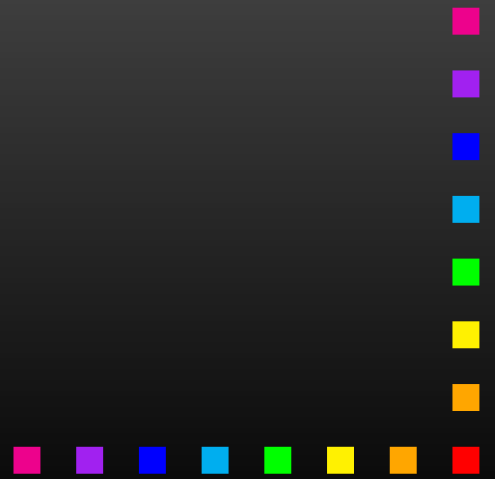
```
file = OpenFortran["bsgamma.F"]

WriteString[file,
  SubroutineDecl["bsgamma(C7,C8)"] <>
  "\tdouble complex C7, C8\n" <>
  "#include \"looptools.h\"\n"]

WriteExpr[file, {C7 -> c7, C8 -> c8}]

WriteString[file, "\tend\n"]

Close[file]
```



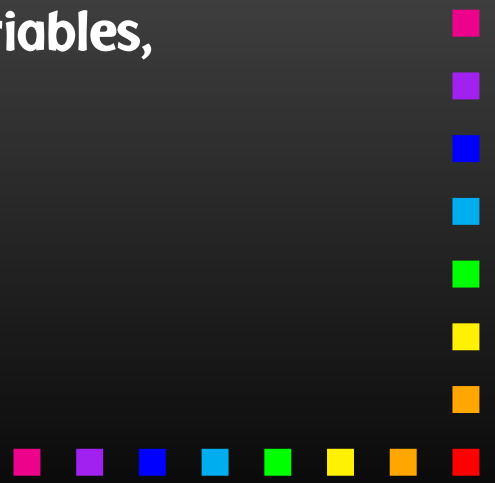
Cuba Parallelization: Design Considerations

No additional software shall be needed.

- OS functions only.
- No parallelization across the network (e.g. via MPI).
- Uses internal cores 'only', thus e.g. 4 or 8.
- Speed-ups not expected to be linear.
- More cores not necessarily useful.

Shall work for any integrand function.

- Requires user's understanding of issues (e.g. global variables, common blocks, I/O buffers).
- Re-coding effort for old code.
- Reentrancy cannot be fully controlled e.g. in Fortran.



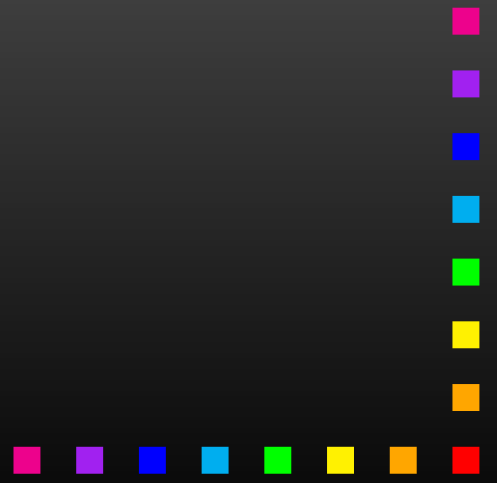
Cuba Parallelization: Design Considerations

Parallelization should work 'automatically.'

- No system knowledge required.
- No re-compile necessary.
- Auto-detect # of cores + load at run-time.
- User control through environment variable `CUBACORES` (Condor).
- Auto-parallelization only acceptable if speed-ups 'reasonable.'

Shall be available on all platforms.

- Native Windows has no `fork` function.
- Cygwin API emulates `fork` but quite slow.
- `fork` is moderately 'expensive' even on Linux/MacOS.
- Keep `fork` calls minimal: 'Spinning Threads' method
  = fork N times at entry into Cuba routine.



Cuba Parallelization: Design Considerations

Usual issues with parallel sample generation.

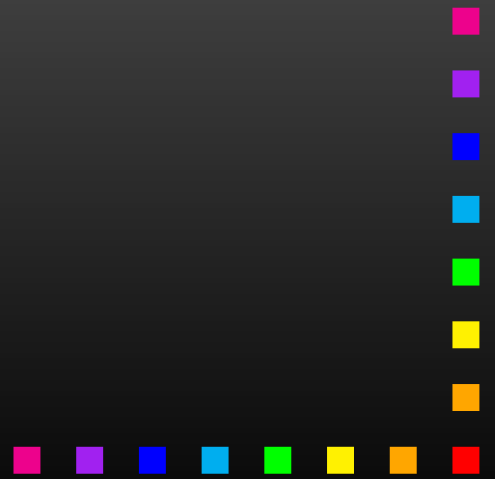
- How to independently seed parallel random-number generators?
- Best to generate samples on master only, distribute to workers.
- 1 Master, N workers on N-core system.

Note: Parallelization as discussed here does not cover Mathematica, where one needs to re-define `MapSample` only, e.g. by `ParallelMap`. (Would need to fork Mathematica kernel, not Cuba executable, license issues etc.)

fork vs. pthread_create

- `pthread_create` creates additional thread in same memory space.
- `fork` creates completely independent process.
- On Linux: pages not actually duplicated until written on ('copy-on-write'), thus no large penalty.

Must use `fork` for non-reentrant integrands.



Master-Worker Communication

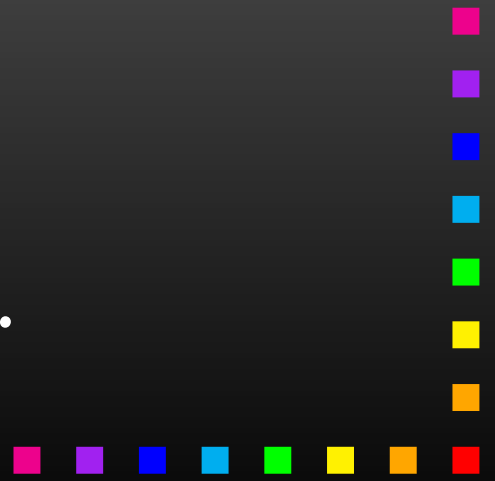
Possible communication channels:

- file read/write,
- pipe read/write,
- socket read/write,
- shared memory (IPC).

I/O creates obvious scheduling point for kernel.
Need semaphore or similar if using shared memory only.

Used in Cuba:

- (if available:) shared memory for samples,
- socketpair read/write for control information.



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.

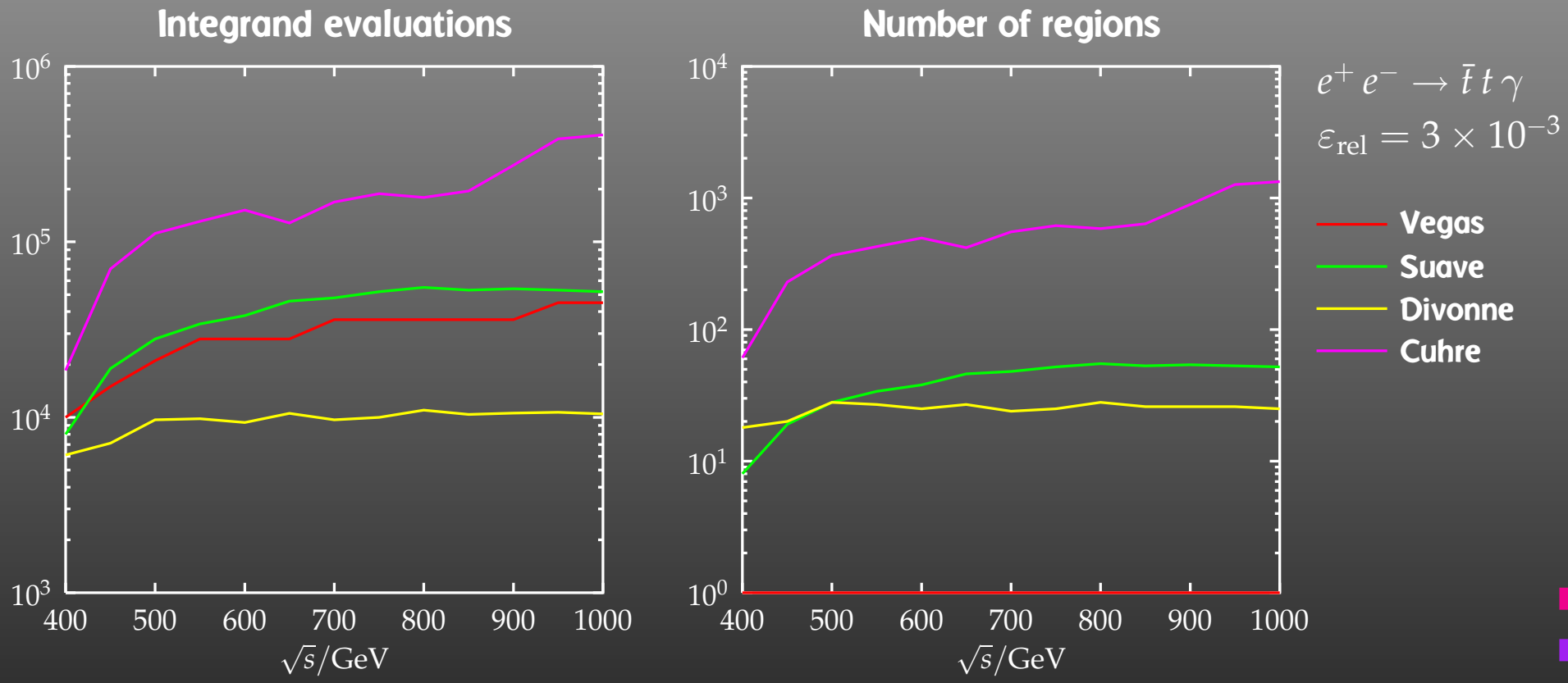
Speed-ups will obviously depend on the 'cost' of the integrand: The more time a single integrand evaluation takes, the better speed-ups can be expected to achieve.

Timing Measurements

Timing measurements delicate on multicore systems:

- System timer (even `ualarm`) has granularity.
- Cannot use timer interrupt directly in integrand delay, accumulates too large errors.
- First calibrate delay loop over sufficiently long time interval.
- Use same calibrated value per machine for all runs.
- Repeat integrations such that each measurement takes a reasonable minimum amount of time (to minimize measurement errors).
- Disable processes like `condor_start`, `autonice`, etc.

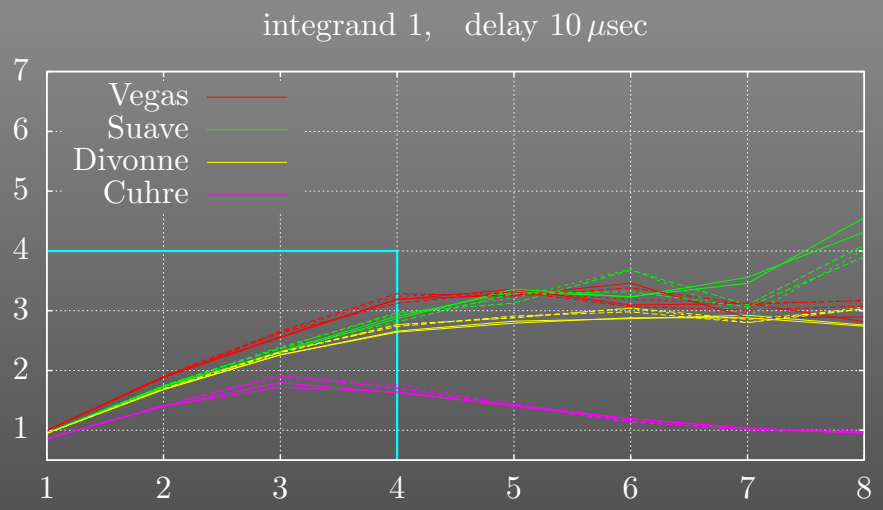
Cuba Comparison

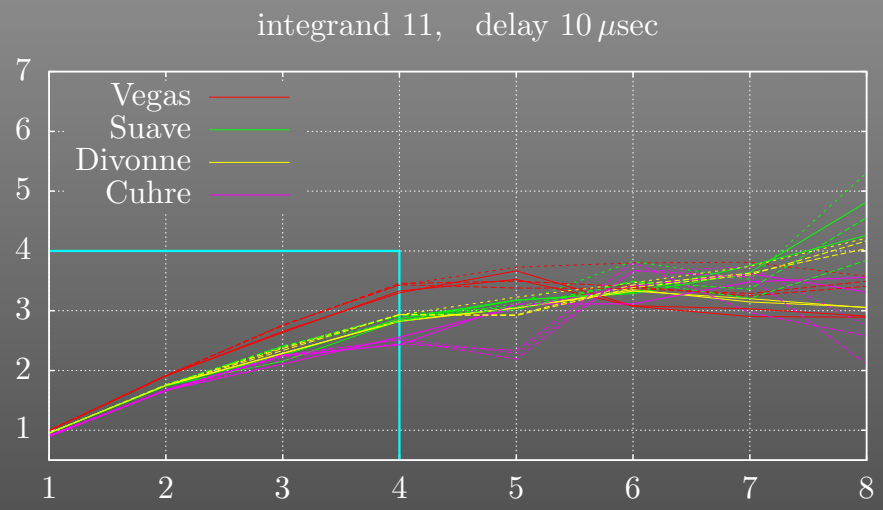


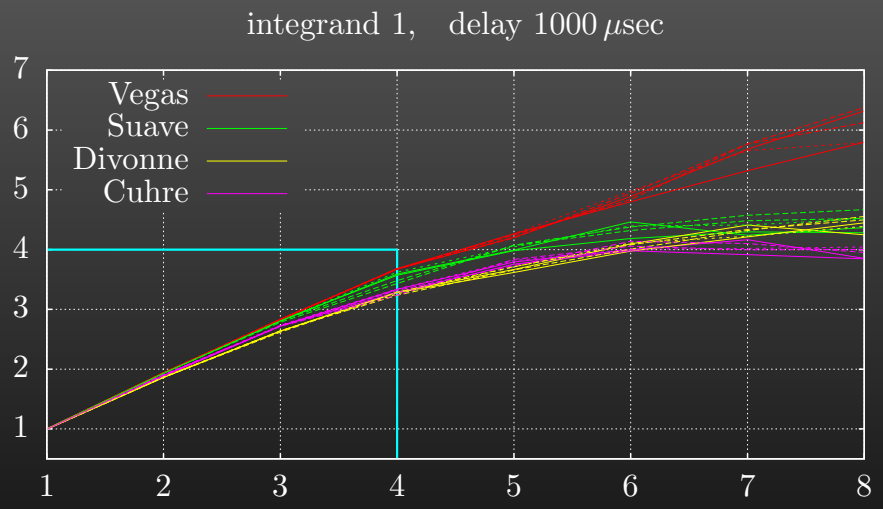
'Gauge' integration problem first:

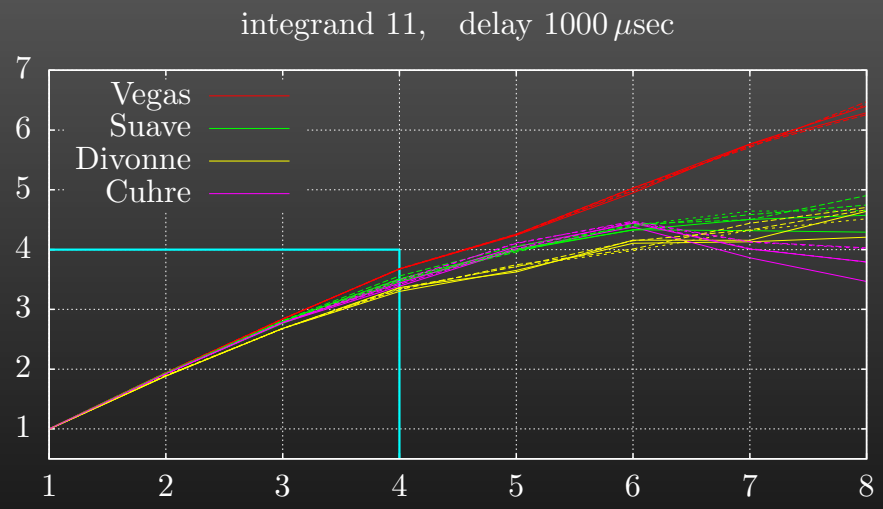
- Compute with all four routines.
- Check whether results are consistent.
- Select fastest algorithm.

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

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

Cuba: feynarts.de/cuba

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