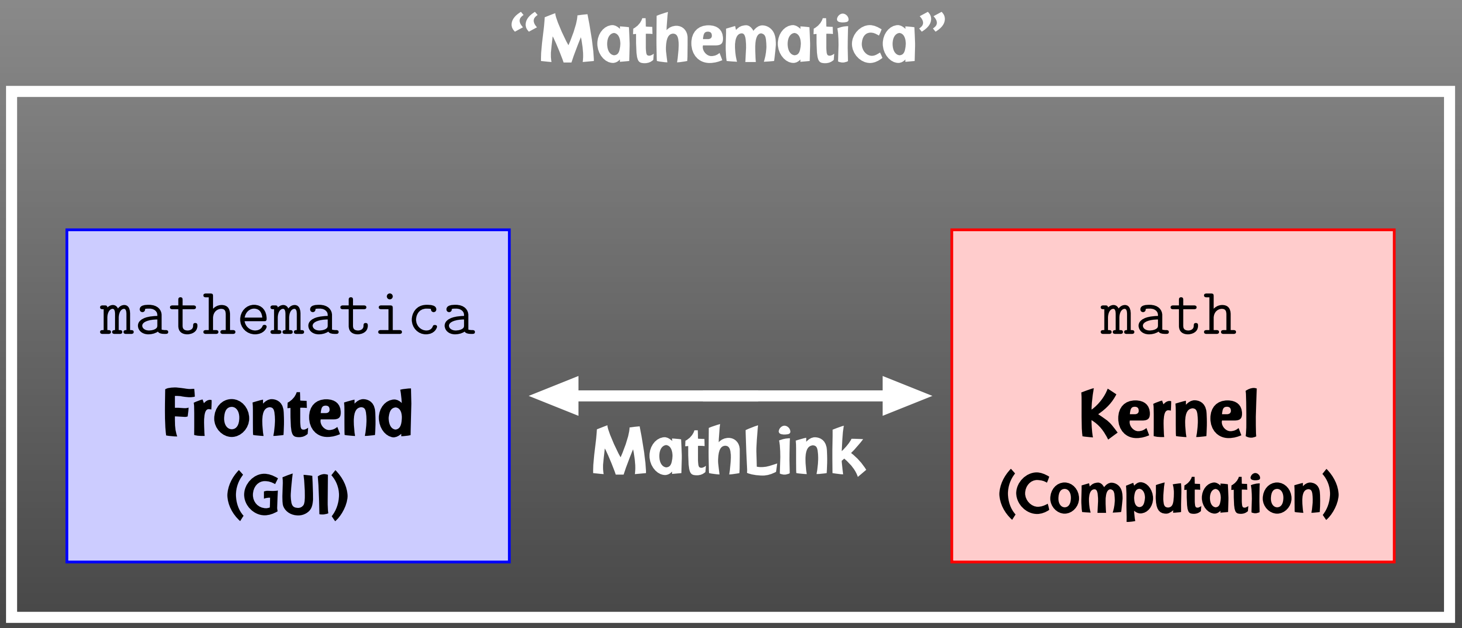
Symbolic Programming in HEP

Thomas Hahn

Max-Planck-Institut für Physik
München

http://wwwth.mpp.mpg.de/members/hahn → **Lecture Material**

Mathematica Components



http://wwwth.mpp.mpg.de/members/hahn/intro_math.pdf

<rant>Why I hate the Frontend</rant>

Frontend:

- ☺ Nice formatting
- ☺ Documentation
- ☺ Ease of use
- ☹ No obvious relation between screen and definitions
- ☹ Always interactive
- ☹ Slow startup

Kernel:

- ☹ Text interface
- ☹ No pretty-printing
- ☺ 1-to-1 relation to definitions
- ☺ Interactive and non-interactive
- ☺ Scriptable
- ☺ Fast startup

Expert Systems

In technical terms, Mathematica is an **Expert System.**
Knowledge is added in form of **Transformation Rules.**
An expression is transformed until no more rules apply.

Example:

```
myAbs[x_] := x /; NonNegative[x]
myAbs[x_] := -x /; Negative[x]
```

We get:

`myAbs[3]` ☞ `3`

`myAbs[-5]` ☞ `5`

`myAbs[2 + 3 I]` ☞ `myAbs[2 + 3 I]`

— no rule for complex arguments so far

`myAbs[x]` ☞ `myAbs[x]`

— no match either

Immediate and Delayed Assignment

Transformations can either be

- added "permanently" in form of Definitions,

```
norm[vec_] := Sqrt[vec . vec]
norm[{1, 0, 2}] ☞ Sqrt[5]
```

- applied once using Rules:

```
a + b + c /. a -> 2 c ☞ b + 3 c
```

Transformations can be Immediate or Delayed. Consider:

```
{r, r} /. r -> Random[] ☞ {0.823919, 0.823919}
{r, r} /. r :> Random[] ☞ {0.356028, 0.100983}
```

Almost everything is a List

All Mathematica objects are either **Atomic**, e.g.

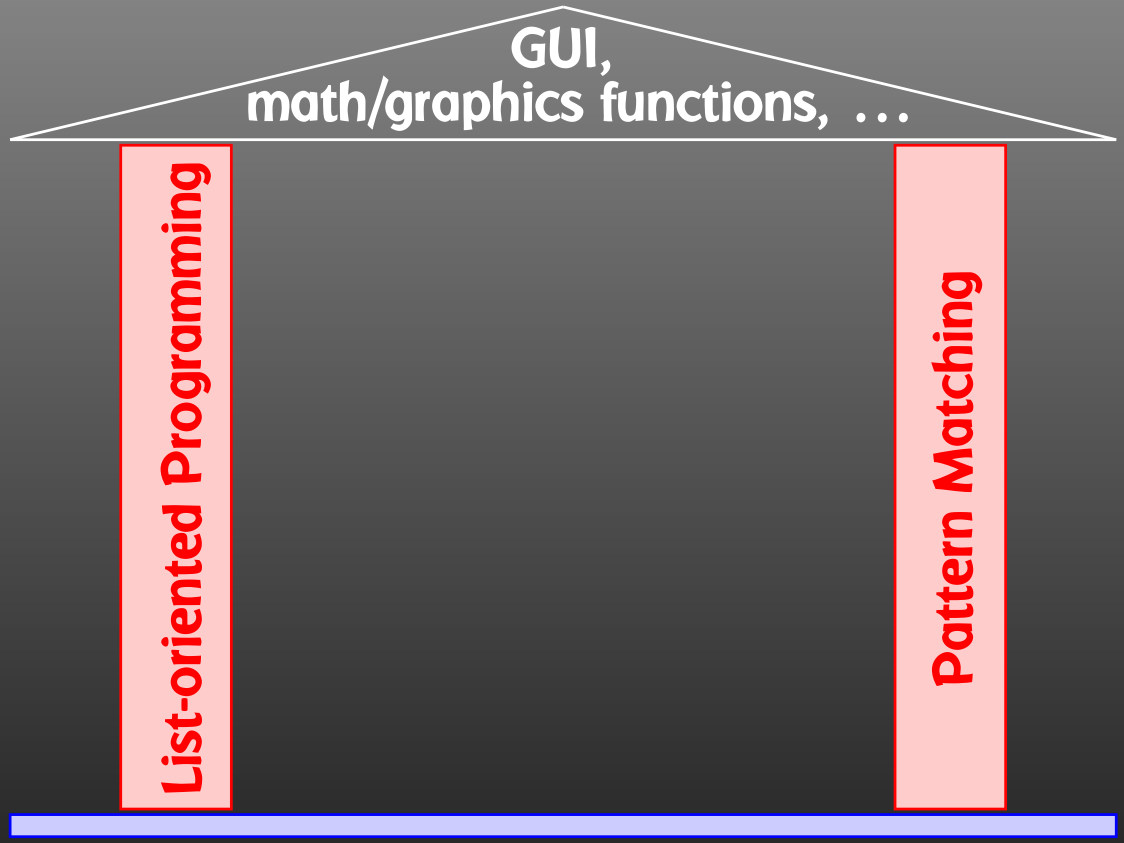
```
Head[133] ☞ Integer
Head[a] ☞ Symbol
```

or (generalized) **Lists** with a **Head** and **Elements**:

```
expr = a + b
```

```
FullForm[expr] ☞ Plus[a, b]
Head[expr] ☞ Plus
expr[[0]] ☞ Plus      — same as Head[expr]
expr[[1]] ☞ a
expr[[2]] ☞ b
```

The Pillars of Mathematica



Map, Apply, and Pure Functions

Map applies a function to all elements of a list:

```
Map[f, {a, b, c}] ☞ {f[a], f[b], f[c]}
f /@ {a, b, c} ☞ {f[a], f[b], f[c]}
```
— short form

Apply exchanges the head of a list:

```
Apply[Plus, {a, b, c}] ☞ a + b + c
Plus @@ {a, b, c} ☞ a + b + c
```
— short form

Pure Functions are a concept from formal logic. A pure function is defined 'on the fly':

```
(# + 1)& /@ {4, 8} ☞ {5, 9}
```

The `#` (same as `#1`) represents the first argument, and the `&` defines everything to its left as the pure function.

List-oriented Programming

Using Mathematica's list-oriented commands is almost always of advantage in both speed and elegance.

Consider:

```
tab = Table[Random[], {10^7}];

test1 := Block[ {sum = 0},
  Do[ sum += tab[[i]], {i, Length[tab]} ];
  sum ]

test2 := Apply[Plus, tab]
```

Here are the timings:

```
Timing[test1][[1]] ☞ 8.29 Second
Timing[test2][[1]] ☞ 1.75 Second
```

List Operations

Flatten removes all sub-lists:

```
Flatten[f[x, f[y], f[f[z]]]] ☞ f[x, y, z]
```

Sort and **Union** sort a list. **Union** also removes duplicates:

```
Sort[{3, 10, 1, 8}] ☞ {1, 3, 8, 10}
Union[{c, c, a, b, a}] ☞ {a, b, c}
```

Prepend and **Append** add elements at the front or back:

```
Prepend[r[a, b], c] ☞ r[c, a, b]
Append[r[a, b], c] ☞ r[a, b, c]
```

Insert and **Delete** insert and delete elements:

```
Insert[h[a, b, c], x, {2}] ☞ h[a, x, b, c]
Delete[h[a, b, c], {2}] ☞ h[a, c]
```

More Speed Bumps

Consider:

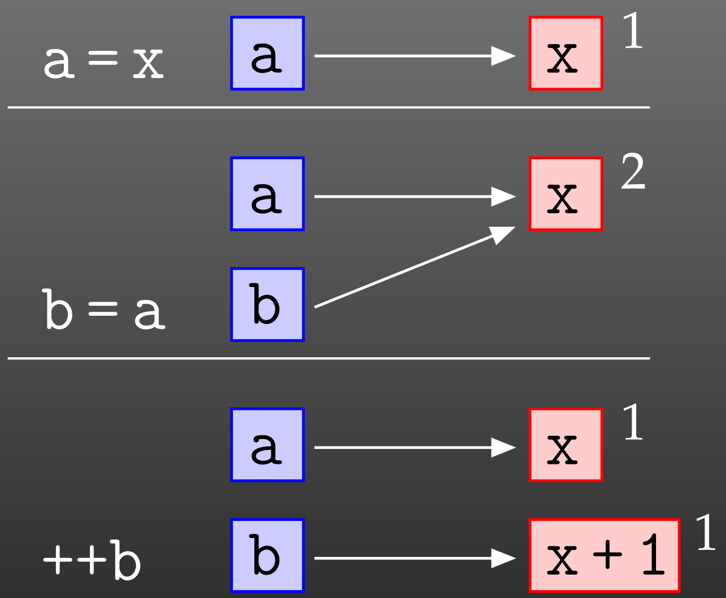
```
tab = Table[Random[], {10^5}];

test1 := Block[ {res = {}},
  Do[ AppendTo[res, tab[[i]]], {i, Length[tab]} ];
  res ]

test2 := Block[ {res = {}},
  Do[ res = {res, tab[[i]]}, {i, Length[tab]} ];
  Flatten[res] ]
```

The timings:

```
Timing[test1][[1]] ☞ 19.47 Second
Timing[test2][[1]] ☞  0.11 Second
```

Reference Count

Assignments that don't change the content make no copy but just increase the Reference Count.



Reference Count and Speed

```
test1 := ...
  ... AppendTo[res, tab[[i]]] ...
  res

test2 :=
  ... res = {res, tab[[i]]} ...
  Flatten[res]
```

`test1` has to **re-write the list every time** an element is added:

{} {1} {1,2} {1,2,3} ...

`test2` does that **only once** at the end with `Flatten`:

{} {{},1} {{{},1},2} {{{{},1},2},3} ...

Patterns

One of the most useful features is Pattern Matching:

`_`	— matches one object
`__`	— matches one or more objects
`___`	— matches zero or more objects
`x_`	— named pattern (for use on the r.h.s.)
`x_h`	— pattern with head `h`
`x_:1`	— default value
`x_?NumberQ`	— conditional pattern
`x_ /; x > 0`	— conditional pattern

Patterns take function overloading to the limit, i.e. functions behave differently depending on *details* of their arguments:

```
Attributes[Pair] = {Orderless}
Pair[p_Plus, j_] := Pair[#, j]& /@ p
Pair[n_?NumberQ i_, j_] := n Pair[i, j]
```

MathLink programming

MathLink is Mathematica's API to interface with C and C++.
J/Link offers similar functionality for Java.

A MathLink program consists of three parts:

a) Declaration Section

```
:Begin:
:Function: mA0
:Pattern: A0[m_, opt___Rule]
:Arguments: {N[m], N[Delta /. {opt} /. Options[A0]],
   N[Mudim /. {opt} /. Options[A0]]}
:ArgumentTypes: {Real, Real, Real}
:ReturnType: Real
:End:

:Evaluate: Options[A0] = {Delta -> 0, Mudim -> 1}
```

MathLink programming

b) C code implementing the exported functions

```
#include "mathlink.h"

static double mA0(const double m,
    const double delta, const double mudim) {
  return (m == 0) ? 0 : m*(1 - log(m/mudim) + delta);
}
```

MathLink programming

c) Boilerplate main function

```
int main(int argc, char **argv) {
  return MLMain(argc, argv);
}
```

Compile with `mcc` instead of `cc`.
Load in Mathematica with `Install["program"]`.

For even more details see arXiv:1107.4379.

Scripting Mathematica

Efficient batch processing with Mathematica:

Put everything into a script, using sh's Here documents:

```
#! /bin/sh .................. Shell Magic
math << \_EOF_ ............ start Here document (note the \)
  AppendTo[$Echo, "stdout"];
  << FeynArts`
  top = CreateTopologies[...];
  ...
_EOF_ ..................... end Here document
```

Everything between "<< *tag*" and "*tag*" goes to Mathematica as if it were typed from the keyboard.

Note the "\\" before *tag*, it makes the shell pass everything literally to Mathematica, without shell substitutions.

Scripting Mathematica

- Everything contained in one compact shell script, even if it involves several Mathematica sessions.
- Can combine with arbitrary shell programming, e.g. can use command-line arguments efficiently:

```
#! /bin/sh
math -run "arg1=$1" -run "arg2=$2" ... << \END
  ...
END
```

- Can easily be run in the background, or combined with utilities such as make.

Debugging hint: `-x` flag makes shell echo every statement,

```
#! /bin/sh -x
```

Commercial Software?

Mathematica licenses cost money ($\sim$ 5 k€/license).

While your Mathematica program runs, it blocks one license, so don't 'just' leave your Mathematica session open.

- Parallelize
- Script, Distribute, Automate
- Crunch numbers outside Mathematica

But: don't overdo it.
If your calculation takes 5 min in total, don't waste time improving.

Parallel Kernels

Mathematica has built-in support for parallel Kernels:

```
LaunchKernels[];
ParallelNeeds["mypackage`"];

data = << mydata;
ParallelMap[myfunc, data];
```

Parallel Kernels count toward Sublicenses.
 # Sublicenses = 8 $\times$ # interactive Licenses.

Parallel Functions

- **More functions:**

  ```
  ParallelArray   ParallelEvaluate  ParallelNeeds
  ParallelSum     ParallelCombine   ParallelTable
  ParallelDo      ParallelProduct   ParallelTry
  ParallelMap     ParallelSubmit
  DistributeDefinitions  DistributeContexts
  ```

- **Automatic parallelization (so-so success):**
  `Parallelize[`*expr*`]`

- **'Intrinsic' functions (e.g. `Simplify`) not parallelizable.**

- **Multithreaded computation partially automatic (OMP) for some numerical functions, e.g. `Eigensystem`.**

- **Take care of side-effects of functions.**

- **Usual concurrency stuff (write to same file, etc).**

Crunch Numbers outside Mathematica

- Conversion of Mathematica expression to Fortran/C painless.
- Optimized output can easily run faster than in Mathematica.
- Showstopper: Functions not available in Fortran/C, e.g. `NDSolve`, `Zeta`. Maybe 3rd-party substitute (GSL, Netlib).
- Mathematica has built-in C-code generator, e.g.

  ```
  myfunc = Compile[{{x}}, x^2 + Sin[x^2]];
  Export["myfunc.c", myfunc, "C"]
  ```

  But no standalone code: shared object for use with Mathematica (i.e. also needs license).
- FormCalc's code-generation functions produce optimized standalone code.

Code-generation Functions

FormCalc's code-generation functions are public and disentangled from the rest of the code. They can be used to **write out an arbitrary Mathematica expression as optimized Fortran or C code:**

- `handle = OpenCode["file.F"]`
  opens *file.F* as a Fortran file for writing,
- `WriteExpr[handle, {var -> expr, ...}]`
  writes out Fortran code which calculates *expr* and stores the result in *var*,
- `Close[handle]`
  closes the file again.

Code generation

Traditionally: Output in Fortran.
Code generator is meanwhile rather sophisticated, 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:
  `VarDecl`, `ToDoLoops`, `IndexIf`, `FileSplit`, ...

C Output

- **Output in C99** makes integration into C/C++ codes easier:

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

Code structured by e.g.

- **Loops and tests handled through macros,** e.g. `LOOP(var,1,10,1)...ENDLOOP(var)`
- Introduced **data types** `RealType` **and** `ComplexType` for better abstraction, can e.g. be changed to different precision.

Mathematica ↔ Fortran

Mathematica → Fortran:

- Get FormCalc from http://feynarts.de/formcalc
- Write out arbitrary Mathematica expression:

```
h = OpenCode["file"]
WriteExpr[h, {var -> expr, ...}]
Close[h]
```

Fortran → Mathematica:

- Get http://feynarts.de/formcalc/FortranGet.tm
- Compile: `mcc -o FortranGet FortranGet.tm`
- Load in Mathematica: `Install["FortranGet"]`
- Read Fortran code: `FortranGet["file.F"]`

Why Fortran?

"I don't know what the programming language of the year 2000 will look like, but I know it will be called Fortran."

– C.A.R. Hoare, ca. 1982

- 'Best' language for number crunching.
- Efficient compilers available (commercial + free).
- Straightforward to link with other languages, e.g. C/C++.

More discussion:

`http://moreisdifferent.com/2015/07/16/why-physicsts-still-use-fortran/`

FORM ↔ Mathematica

Mathematica → FORM:

- Get FormCalc from http://feynarts.de/formcalc
- After compilation the `ToForm` utility should be in the executables directory (e.g. Linux-x86-64):

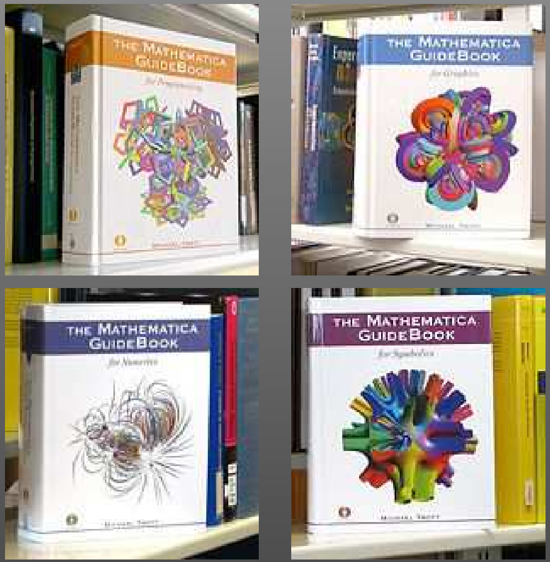
  `ToForm < file.m > file.frm`

FORM → Mathematica:

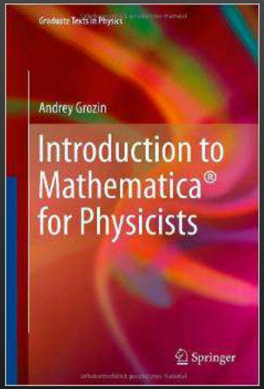
- Get http://feynarts.de/formcalc/FormGet.tm
- Compile it with `mcc -o FormGet FormGet.tm`
- Load it in Mathematica with `Install["FormGet"]`
- Read a FORM output file: `FormGet["file.out"]`
  Pipe output from FORM: `FormGet["!form file.frm"]`

Books

- **Michael Trott**
  The Mathematica Guidebook for $\{$Programming, Graphics, Numerics, Symbolics$\}$ (4 vol)
  Springer, 2004–2006.



- **Andrei Grozin**
  Introduction to Mathematica for Physicists
  Springer, 2013.

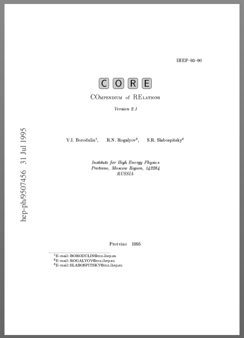


List of Examples

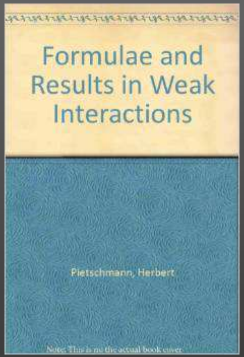
- **Antisymmetric Tensor**
  Built-in in FORM, easy in Mathematica.
- **Application of Momentum Conservation**
  Easy in Mathematica, complicated in FORM.
- **Abbreviationing**
  Easy in Mathematica, new in FORM.
- **Simplification of Color Structures**
  Different approaches.
- **Calculation of a Fermion Trace**
  Built-in in FORM, complicated in Mathematica.
- **Tensor Reduction**

Reference Books, Formula Collections

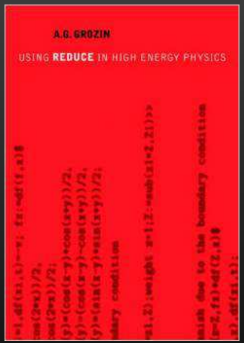
- V.I. Borodulin et al.
  CORE (Compendium of Relations)
  hep-ph/9507456 (v2), arXiv:1702.08246 (v3).



- Herbert Pietschmann
  Formulae and Results in Weak Interactions
  Springer (Austria) 2nd ed., 1983.



- Andrei Grozin
  Using REDUCE in High-Energy Physics
  Cambridge University Press, 1997.



Antisymmetric Tensor

The **Antisymmetric Tensor in n dimensions** is denoted by $\varepsilon_{i_1 i_2 \dots i_n}$. You can think of it as a matrix-like object which has either -1, 0, or 1 at each position.

For example, the **Determinant** of a matrix, being a **completely antisymmetric** object, can be written with the ε-tensor:

$$\det A = \sum_{i_1,\dots,i_n=1}^{n} \varepsilon_{i_1 i_2 \dots i_n} A_{i_1 1} A_{i_2 2} \cdots A_{i_n n}$$

In practice, the ε-tensor is usually contracted, e.g. with vectors. We will adopt the following notation to avoid dummy indices:

$$\varepsilon_{\mu\nu\rho\sigma} p^\mu q^\nu r^\rho s^\sigma = \varepsilon(p, q, r, s)\,.$$

Antisymmetric Tensor in Mathematica

```
Eps[___, p_, ___, p_, ___] := 0

   (* implement linearity: *)

Eps[a___, p_Plus, b___] := Eps[a, #, b]&/@ p

Eps[a___, n_?NumberQ r_, b___] := n Eps[a, r, b]

   (* otherwise sort the arguments into canonical order: *)

Eps[args__] := Signature[{args}] Eps@@ Sort[{args}] /;
  !OrderedQ[{args}]
```

Momentum Conservation

Problem: **Proliferation of terms** in expressions such as

$$d = \frac{1}{(p_1 + p_2 - p_3)^2 + m^2}$$
$$= \frac{1}{p_1^2 + p_2^2 + p_3^2 + 2p_1 p_2 - 2p_2 p_3 - 2p_1 p_3 + m^2},$$

whereas if $p_1 + p_2 = p_3 + p_4$ we could have instead

$$d = \frac{1}{p_4^2 + m^2}.$$

In Mathematica: just do `d /. p1 + p2 - p3 -> p4`.
Problem: FORM cannot replace sums.

Momentum Conservation in FORM

Idea: for each expression x, add and subtract a zero, i.e. form

$$\{x, y = x + 0, z = x - 0\}, \text{ where e.g. } 0 = p_1 + p_2 - p_3 - p_4\,,$$

then select the shortest expression. But: how to select the shortest expression (in FORM)?

Solution: add the number of terms of each argument, i.e.

$$\{x, y, z\} \to \{\overset{1}{x}, \overset{2}{y}, \overset{3}{z}, \overset{4}{n_x}, \overset{5}{n_y}, \overset{6}{n_z}\}\,.$$

Then sort n_x, n_y, n_z, but when exchanging n_a and n_b, **exchange also** a and b:

```
symm 'foo' (4,1) (5,2) (6,3);
```

This unconventional sort statement is rather typical for FORM.

Momentum Conservation in FORM

```
#procedure Shortest(foo)

id `foo'([x]?) = `foo'([x], [x] + `MomSum', [x] - `MomSum');

* add number-of-terms arguments
id `foo'([x]?, [y]?, [z]?) = `foo'([x], [y], [z],
  nterms_([x]), nterms_([y]), nterms_([z]) );

* order according to the nterms
symm `foo' (4,1) (5,2) (6,3);

* choose shortest argument
id `foo'([x]?, ?a) = `foo'([x]);

#endprocedure
```

Abbreviationing

One of the most powerful tricks to both reduce the size of an expression and reveal its structure is to substitute subexpressions by new variables.

The essential function here is `Unique` with which new symbols are introduced. For example,

```
Unique["test"]
```

generates e.g. the symbol `test1`, which is guaranteed not to be in use so far.

The `Module` function which implements lexical scoping in fact uses `Unique` to rename the symbols internally because Mathematica can really do dynamical scoping only.

Abbreviationing in Mathematica

```
$AbbrPrefix = "c"

abbr[expr_] := abbr[expr] = Unique[$AbbrPrefix]

   (* abbreviate function *)
Structure[expr_, x_] := Collect[expr, x, abbr]

   (* get list of abbreviations *)
AbbrList[] := Cases[DownValues[abbr],
  _[_[_[f_]], s_Symbol] -> s -> f]

   (* restore full expression *)
Restore[expr_] := expr /. AbbrList[]
```

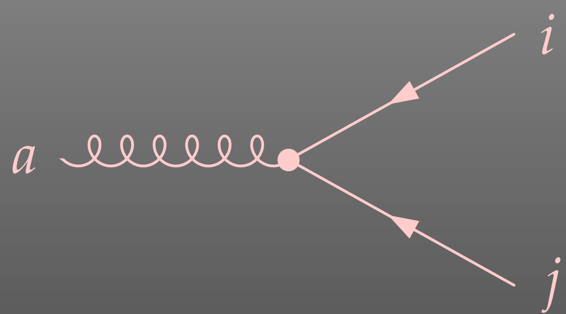
Abbreviationing in FORM

```
* collect w.r.t. some function

b Den;
.sort
collect acc;

* introduce abbreviations for prefactors

toPolynomial onlyfunctions acc;
.sort

* print abbreviations & abbreviated expr

#write "%X"
print +s;
```

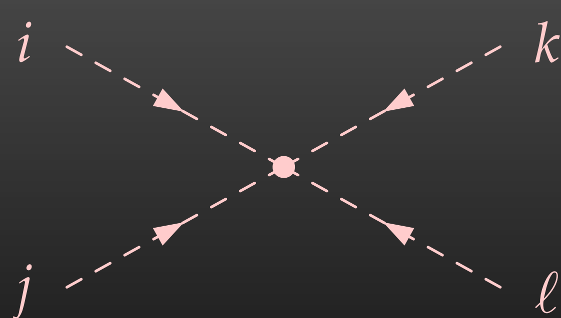
Color Structures

In Feynman diagrams four types of Color structures appear:

Natural Representation

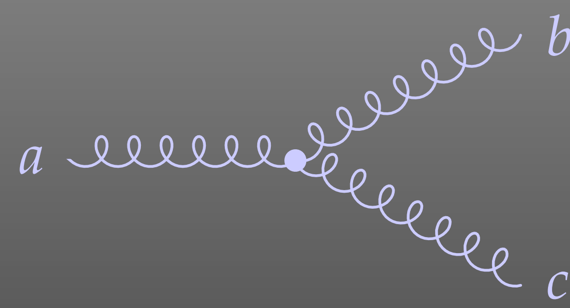


$\sim T^a_{ij}$ =SUNT[a,i,j]

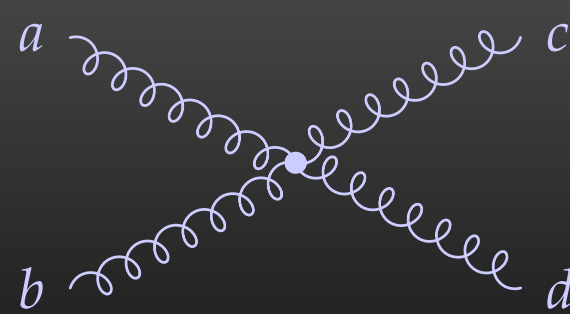


$\sim T^a_{ij} T^a_{k\ell}$ = SUNTSum[i,j,k,ℓ]

Adjoint Representation



$\sim f^{abc}$ = SUNF[a,b,c]



$\sim f^{abx} f^{xcd}$ = SUNF[a,b,c,d]

Unified Notation

The `SUNF`'s can be converted to `SUNT`'s via

$$f^{abc} = 2\mathrm{i}\big[\mathrm{Tr}(T^c T^b T^a) - \mathrm{Tr}(T^a T^b T^c)\big]\,.$$

We can now represent all color objects by just `SUNT`:

- `SUNT`$[i, j] = \delta_{ij}$
- `SUNT`$[a, b, \ldots, i, j] = (T^a T^b \cdots)_{ij}$
- `SUNT`$[a, b, \ldots, 0, 0] = \mathrm{Tr}(T^a T^b \cdots)$

This notation again avoids unnecessary dummy indices.
(Mainly namespace problem.)

For purposes such as the "large-N_c limit" people like to use SU(N) rather than an explicit SU(3).

Fierz Identities

The **Fierz Identities** relate expressions with **different orderings of external particles.** The Fierz identities essentially express completeness of the underlying matrix space.

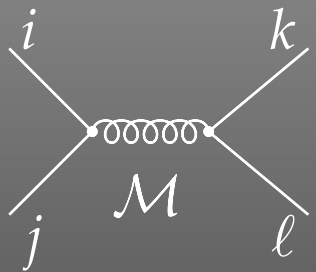
They were originally found by Markus Fierz in the context of Dirac spinors, but can be generalized to any finite-dimensional matrix space [hep-ph/0412245].

For SU(N) (color) reordering, we need

$$T^a_{ij} T^a_{k\ell} = \frac{1}{2} \left(\delta_{i\ell} \delta_{kj} - \frac{1}{N} \delta_{ij} \delta_{k\ell} \right) .$$

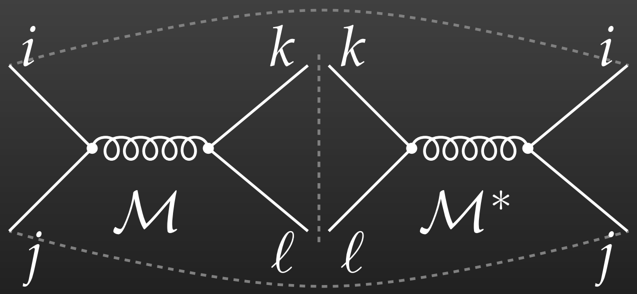
Cvitanovich Algorithm

For an **Amplitude:**



- convert all color structures to (generalized) `SUNT` objects,
- simplify: apply Fierz identity on all internal gluon lines,
- expect `SUNT` with indices of external particles to remain.

For a **Squared Amplitude:**



- use the Fierz identity to get rid of all `SUNT` objects,
- expect `SUNT` to vanish, color factors (numbers) only.

For “hand” calculations, a pictorial version of this algorithm exists in the literature.

Color Simplify in FORM

```
* introduce dummy indices for the traces
repeat;
  once SUNT(?a, 0, 0) = SUNT(?a, DUMMY, DUMMY);
  sum DUMMY;
endrepeat;

* take apart SUNTs with more than one T
repeat;
  once SUNT(?a, [a]?, [b]?, [i]?, [j]?) =
    SUNT(?a, [a], [i], DUMMY) * SUNT([b], DUMMY, [j]);
  sum DUMMY;
endrepeat;

* apply the Fierz identity
id SUNT([a]?, [i]?, [j]?) * SUNT([a]?, [k]?, [l]?) =
  1/2 * SUNT([i], [l]) * SUNT([j], [k]) -
  1/2/(`SUNN') * SUNT([i], [j]) * SUNT([k], [l]);
```

Translation to Color-Chain Notation

In color-chain notation we can distinguish two cases:

a) Contraction of different chains:

$$\langle A|\, T^a \,|B\rangle \, \langle C|\, T^a \,|D\rangle = \frac{1}{2}\Big(\langle A|D\rangle \, \langle C|B\rangle - \frac{1}{N}\, \langle A|B\rangle \, \langle C|D\rangle\Big)\,,$$

b) Contraction on the same chain:

$$\langle A|\, T^a \,|B|\, T^a \,|C\rangle = \frac{1}{2}\Big(\langle A|C\rangle \,\mathrm{Tr}\, B - \frac{1}{N}\, \langle A|\, B \,|C\rangle\Big)\,.$$

Color Simplify in Mathematica

```
   (* same-chain version *)
sunT[t1___, a_Symbol, t2___, a_, t3___, i_, j_] :=
  (sunT[t1, t3, i, j] sunTrace[t2] -
    sunT[t1, t2, t3, i, j]/SUNN)/2

   (* different-chain version *)
sunT[t1___, a_Symbol, t2___, i_, j_] *
sunT[t3___, a_, t4___, k_, l_] ^:=
  (sunT[t1, t4, i, l] sunT[t3, t2, k, j] -
    sunT[t1, t2, i, j] sunT[t3, t4, k, l]/SUNN)/2

   (* introduce dummy indices for the traces *)
sunTrace[a__] := sunT[a, #, #]&[ Unique["col"] ]
```

Fermion Trace

Leaving apart problems due to γ_5 in d dimensions, we have as the main algorithm for the 4d case:

$$\begin{aligned}\mathrm{Tr}\,\gamma_\mu\gamma_\nu\gamma_\rho\gamma_\sigma\cdots = \; & + g_{\mu\nu}\,\mathrm{Tr}\,\gamma_\rho\gamma_\sigma\cdots \\ & - g_{\mu\rho}\,\mathrm{Tr}\,\gamma_\nu\gamma_\sigma\cdots \\ & + g_{\mu\sigma}\,\mathrm{Tr}\,\gamma_\nu\gamma_\rho\cdots\end{aligned}$$

This algorithm is recursive in nature, and we are ultimately left with

$$\mathrm{Tr}\,\mathbb{1} = 4\,.$$

(Note that this 4 is not the space-time dimension, but the dimension of spinor space.)

Fermion Trace in Mathematica

```
Trace4[mu_, g__] :=
Block[ {Trace4, s = -1},
  Plus@@ MapIndexed[
    ((s = -s) Pair[mu, #1] Drop[Trace4[g], #2])&,
    {g} ]
]

Trace4[] = 4
```

Tensor Reduction

The loop integrals corresponding to closed loops in a Feynman integral in general have a **tensor structure** due to **integration momenta in the numerator.** For example,

$$B_{\mu\nu}(p) = \int \mathrm{d}^d q \, \frac{q_\mu q_\nu}{\left(q^2 - m_1^2\right)\left((q-p)^2 - m_2^2\right)} \,.$$

Such tensorial integrals are rather unwieldy in practice, therefore they are reduced to linear combinations of Lorentz-covariant tensors, e.g.

$$B_{\mu\nu}(p) = B_{00}(p)\, g_{\mu\nu} + B_{11}(p)\, p_\mu p_\nu \,.$$

It is the **coefficient functions** B_{00} and B_{11} which are implemented in a library like LoopTools.

Tensor Reduction Algorithm

The first step is to convert the integration momenta in the numerator to an actual tensor, e.g. $q_\mu q_\nu \to N_{\mu\nu}$. FORM has the special command `totensor` for this:

```
totensor q1, NUM;
```

The next step is to take out $g_{\mu\nu}$'s in all possible ways. We do this in form of a sum:

$$N_{\mu_1\dots\mu_n} = \sum_{i=0,2,4,\dots}^{n} \pi(0)^i \sum_{\substack{\text{all } \{\nu_1,\dots,\nu_i\} \\ \in\{\mu_1,\dots,\mu_n\}}} g_{\nu_1\nu_2}\cdots g_{\nu_{i-1}\nu_i} N_{\mu_1\dots\mu_n\setminus\nu_1\dots\nu_i}$$

The $\pi(0)^i$ keeps track of the indices of the tensor coefficients, i.e. it later provides the two zeros for every $g_{\mu\nu}$ in the index, as in D_{0012}.

Tensor Reduction Algorithm

To fill in the remaining $\pi(i)$'s, we start off by tagging the arguments of the loop function, which are just the momenta. For example:

$$C(p_1, p_2, \ldots) \to \tau\big(\pi(1)p_1 + \pi(2)p_2\big)\, C(p_1, p_2, \ldots)$$

The temporary function τ keeps its argument, the 'tagged' momentum p, separate from the rest of the amplitude.

Now add the indices of $N_{\mu_1\ldots\mu_n}$ to the momentum in τ:

$$\tau(p)\, N_{\mu_i\ldots\mu_n} = p_{\mu_i}\cdots p_{\mu_n}\,.$$

Finally, collect all π's into the tensor-coefficient index.

Tensor Reduction in FORM

```
totensor q1, NUM;

* take out 0, 2, 4... indices for g_{mu nu}
id NUM(?b) = sum_(DUMMY, 0, nargs_(?b), 2,
  pave(0)^DUMMY * distrib_(1, DUMMY, dd_, NUM, ?b));

* construct tagged momentum in TMP
id C0i([p1]?, [p2]?, ?a) = TMP(pave(1)*[p1] + pave(2)*[p2]) *
  C0i(MOM([p1]), MOM([p2] - [p1]), MOM([p2]), ?a);

* expand momentum
repeat id TMP([p1]?) * NUM([mu]?, ?a) =
  d_([p1], [mu]) * NUM(?a) * TMP([p1]);

* collect the indices
chainin pave;
```

Tensor Reduction in Mathematica

```
tens[i_, p_][mu_, nu___] :=
Block[ {tens},
        (* take out g *)
  { MapIndexed[g[mu, #1] Drop[tens[{i,0,0}, p][nu], #2]&, {nu}],
        (* take out p *)
    (#1[mu] tens[{i,#2}, p][nu])&@@@ p }
]

tens[i_, _][] := C@@ Sort[Flatten[i]]



FindTensors[mu_, p_] :=
Block[ {tenslist},
  tenslist = tens[{}, MapIndexed[List, p]]@@ mu;
  Collect[Plus@@ Flatten[tenslist], _C]
]
```

More Complex Calculations

Often special requirements:

- **Resummations** (e.g. hbb in MSSM),
- **Approximations** (e.g. gaugeless limit),
- **K-factors**,
- Nontrivial **renormalization.**

Software design so far:

- Mostly **'monolithic'** (one package does everything).
- Often controlled by **parameter cards**, not easy to use beyond intended purpose.
- May want to/must use other packages.

Example: $\mathcal{O}(\alpha_t^2)$ MSSM Higgs-mass corrections

Hollik, Paßehr 2014

Shopping List for the Diagrammatic Calculation:

① **Unrenormalized 2L self-energies**
$\Sigma^{(2)}_{hh}$, $\Sigma^{(2)}_{hH}$, $\Sigma^{(2)}_{hA}$, $\Sigma^{(2)}_{HH}$, $\Sigma^{(2)}_{HA}$, $\Sigma^{(2)}_{AA}$, $\Sigma^{(2)}_{H^+H^-}$
in gaugeless approximation at $p^2 = 0$ at $\mathcal{O}(\alpha_t^2)$.

② **1L diagrams with insertions of 1L counterterms.**

③ **2L counterterms for ①.**

④ **2L tadpoles $T^{(2)}_h$, $T^{(2)}_H$, $T^{(2)}_A$ at $\mathcal{O}(\alpha_t^2)$ appearing in ③.**

Template for Calculations

- Break calculation into several steps.
- Implement each step as independent program (invoked from command line).
- In lieu of ‘in vivo’ debugging keep detailed logs.
- Coordinate everything through a makefile.

Steps of the Calculation

Calculation split into 7 (8) steps:

FeynArts ⇒ `1-amps` → `2-prep` → `3-calc` ⇐ TwoCalc ⇐ FormCalc

`1-amps`: diagram generation

`2-prep`: preparation for tensor reduction

`3-calc`: tensor reduction

`0-glmod` ⇐ MSSMCT.mod

`0-glmod`: model file preparation ↑ `1-amps`

`3-calc` ↓ `4-simp`

`4-simp`: simplification

`4-simp` ↓ `5-rc`

FormCalc ⇒ `7-code` ← `6-comb` ← `5-rc` ⇐ FormCalc

`7-code`: code generation

`6-comb`: combination of results

`5-rc`: calculation of renorm. constants

Script Structure

- Shell scripts (`/bin/sh`), run from command line as e.g.
  `./1-amps arg1 arg2`
- `arg1` = `h0h0`, `h0HH`, `h0A0`, `HHHH`, `HHA0`, `A0A0`, `HmHp` (self-energies),
  `h0`, `HH`, `A0` (tadpoles).
- `arg2` = `0` for virtual 2L diagrams,
  `1` for 1L diagrams with 1L counterterms.
- Inputs/outputs defined in first few lines, e.g.

```
in=m/$1/2-prep.$2
out=m/$1/3-calc.$2
```

- Symbolic output + log files go to '`m`' subdirectory.
  Log file = Output file + `.log.gz`
- Fortran code goes to '`f`' subdirectory.

Step 0: Gaugeless Limit

Gaugeless approximation:

① Set gauge couplings $g, g' = 0 \Rightarrow M_W, M_Z = 0$.

② Keep finite weak mixing angle.

③ Keep $\dfrac{\delta M_W^2}{M_W^2}$ and $\dfrac{\delta M_Z^2}{M_Z^2}$ finite.

Must set $m_b = 0$ so that $\mathcal{O}(\alpha_t^2)$ corrections form supersymmetric and gauge-invariant subset.

Most efficient to modify Feynman rules (not ③, though):

- Load `MSSMCT.mod` model file.
- Modify couplings, remove zero ones.
- Write out `MSSMCTgl.mod` model file.

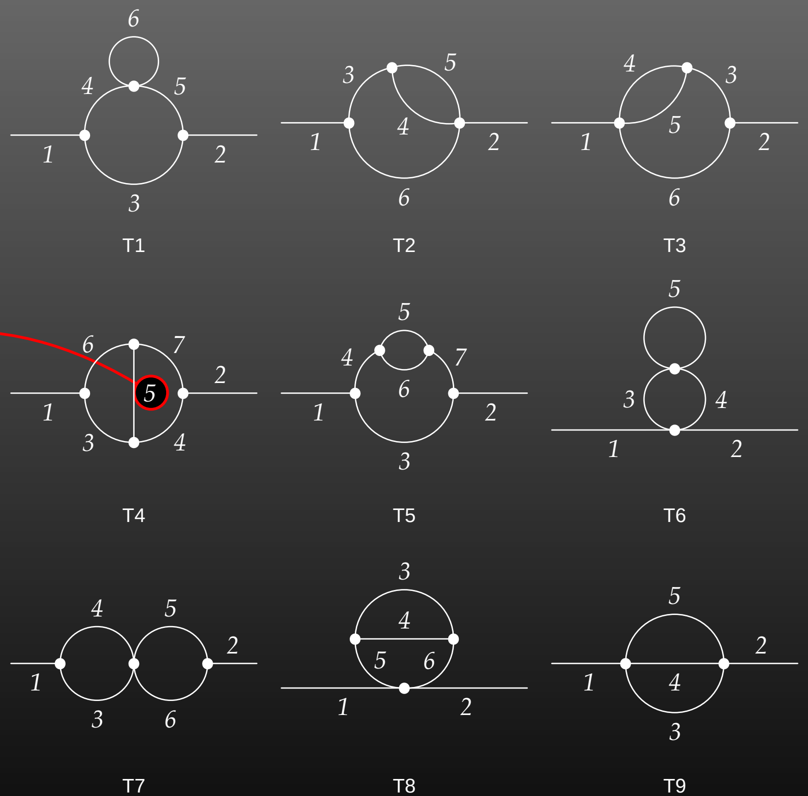
Step 1: Diagram Generation

- **Generate 2L virtual and 1L+counterterm diagrams using wrappers for FeynArts functions.**

Simple diagram selection functions, e.g.

```
sel[0][S[_] -> S[_]] = {
  t[3] && htb[6],
  t[3] && tb[6],
  t[3] && tb[6],
  t[3] && t[4] && htb[5],
  t[3] && htb[5|6],
  t[3] && htb[5],
  t[3] && t[5],
  t[5] && ht[3|4],
  t[3|4|5] && ht[3|4|5] }
```

one of $h_i, \tilde{\chi}, t, \tilde{t}, b, \tilde{b}$



Step 2: Preparation for Tensor Reduction

- Take $p^2 \to 0$ limit.
- Simplify ubiquitous sfermion mixing matrices U_{ij}, mostly by exploiting unitarity ($\sim$ 50% size reduction).

Efficiently Exploit Unitarity in Mathematica

Unitarity of 2 x 2 matrix: $UU^\dagger = U^\dagger U = \mathbb{1}$, **i.e.**

$$
\begin{aligned}
U_{11}U_{11}^* + U_{12}U_{12}^* &= 1, & U_{11}U_{21}^* + U_{12}U_{22}^* &= 0, \\
U_{21}U_{21}^* + U_{22}U_{22}^* &= 1, & U_{21}U_{11}^* + U_{22}U_{12}^* &= 0, \\
U_{11}U_{11}^* + U_{21}U_{21}^* &= 1, & U_{11}U_{12}^* + U_{21}U_{22}^* &= 0, \\
U_{12}U_{12}^* + U_{22}U_{22}^* &= 1, & U_{12}U_{11}^* + U_{22}U_{21}^* &= 0.
\end{aligned}
$$

Problem: `Simplify` **will rarely arrange the U's in just the way that these rules can be applied directly.**

Solution: Introduce auxiliary symbols which immediately deliver the r.h.s. once `Simplify` **considers the l.h.s., i.e. increase the 'incentive' for** `Simplify` **to use the r.h.s.**

But: Upvalues work only one level deep.

Efficiently Exploit Unitarity in Mathematica

Introduce

$\texttt{USf[1,}j\texttt{] USfC[1,}j\texttt{]} \longrightarrow \texttt{UCSf[1,}j\texttt{]}$,

$\texttt{USf[2,}j\texttt{] USfC[2,}j\texttt{]} \longrightarrow \texttt{UCSf[2,}j\texttt{]}$,

$\texttt{USf[1,}j\texttt{] USfC[2,}j\texttt{]} \longrightarrow \texttt{UCSf[3,}j\texttt{]}$, **+ ditto for 1st index**

and formulate unitarity for the `UCSf`:

```
UCSf[2,1] = UCSf[1,2];    UCSf[3,2] = -UCSf[3,1];
UCSf[2,2] = UCSf[1,1];    UCSfC[3,2] = -UCSfC[3,1];
                          UCSf[2,3] = -UCSf[1,3];
...                       UCSfC[2,3] = -UCSfC[1,3];
```

Step 3: Tensor Reduction

- Relatively straightforward application of TwoCalc and FormCalc for tensor reduction.
- Observe: Need two Mathematica sessions since TwoCalc and FormCalc cannot be loaded into one session, easily accomodated in shell script.

Step 4: Simplification

- Tensor reduction traditionally increases # of terms most.
- Step 4 reduces size before combination of results.
- Empirical simplification recipe.
- ‘DiagMark’ trick (D. Stöckinger):
  - Introduce `DiagMark`$[m_i]$ where m_i = masses in loop in FeynArts output.
  - Few simplifications can be made between parts with different `DiagMark` $\Rightarrow$ Can apply simplification as

    ```
    Collect[amp, _DiagMark, simpfunc]
    ```
  - Much faster.

Step 5: Calculation of Renormalization Constants

- Compute 1L renormalization constants (RC) with FormCalc.
- Substitute explicit mass dependence in

$$\texttt{dMVsq1} \to \texttt{MV2}\ \texttt{dMVsq1MV2} \qquad (V = W, Z)$$

such that gaugeless limit can be taken safely.
- Expand in ε, collect powers for easier handling later, e.g.

```
{ dMf1[3,3] -> RC[-1, dMf1[-1,3,3]] +
               RC[0, dMf1[0,3,3]],          expansion
  {dMf1[-1,3,3] -> ...,                     actual expressions for ε-coeffs
   dMf1[0,3,3] -> ...}}
```

Step 6: Combination of Results

- Expand amplitude in ε (similar as RC).
- Insert RCs.
- Add genuine 2L counterterms (hand-coded).
- Pick only ε^0 term (unless debug flag set).
- Perform final simplification.

Step 7: Code Generation

- Introduce abbreviations to shorten code.
- Write out Fortran code using FormCalc's code-generation functions.
- Add static code which computes e.g. the necessary parameters for the generated code.
- Total final code size: 350 kBytes.

More details in arXiv:1508.00562.

Summary

- Mathematica makes it easy, even for fairly unskilled users, to manipulate expressions.
- Mathematica is a general-purpose system, i.e. convenient to use, but not ideal for everything.
- Take advantage of many packages, convert if necessary.
- Scripting helps combine different packages.
- Crunch numbers outside of Mathematica.