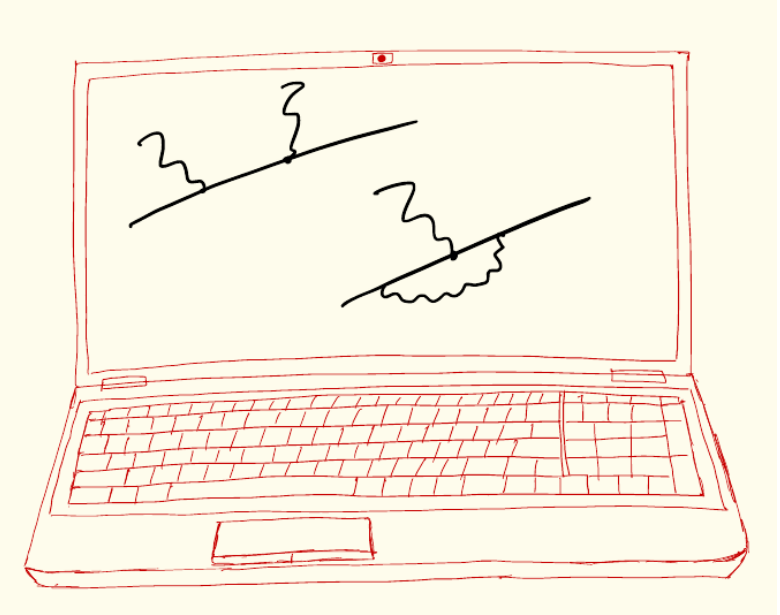
Computer Algebra for Feynman Graphs



7.

In this Lecture

- Detailed description of some useful commands and statements related to:
  - Settings control at runtime
  - Manipulations with functions
  - Manipulations with terms in expressions and $-variables

Useful Commands

- Here we provide more detailed information about a few useful compiler statements (some of which we already considered), grouped by types depending on what these statements do
- General statements
  - `dimension`
- Settings control statements
  - `on/off`
- Manipulations with functions
  - `symmetrize` (`antisymmetrize`, `(r)cyclesymmetrize`)
  - `argument/endargument`
  - `chainin/chainout`
  - `commuteinset`
  - `transform`

Useful Commands

- Manipulations with expressions and terms
  - `global/local`
  - `drop/ndrop`
  - `skip/nskip`
  - `hide/unhide/nhide/nunhide/intohide`
  - `inside/endinside`
  - `term/endterm, sort`
  - `multiply`
  - `discard`

Dimension

- `dimension 5;` Declares the current default dimension to be a number
- `dimension D;` The same as above but the dimension is a symbol (`D` will be implicitly declared as such if not already declared)
- `dimension D:[D-4];` In addition to a symbolic dimension, the declaration can include the name of a symbol that serves as the dimension minus 4.
- The declaration of dimension is used in various contraction identitites that involve built-in tensors such as `d_`, `e_`, and `g_`
- The default dimension in FORM is 4
- These options can also be used in the declaration of indices, such as `index i=6, m=D, n=D:[D-4];`

[Example file: dimension.frm]

On/Off

- The statements `on` and `off` provide means of control the settings of FORM during the execution of the program by switching on or off a certain setup parameter [Example file: on_off.frm]
- The syntax is either of these statements followed by an option, e.g.,

```
on statistics
on names
off finalstats
```

- The scope of each statement comprises the current module and all subsequent modules until either a `.clear` instruction or a new `on/off` statement that overrides previous settings is encountered
- Some of the useful options [and their default values] are the following (you are referred to the reference manual for the rest):
  - `[off] names:` print the names of defined variables and expressions
  - `[off] allnames:` as `names` plus prints the names of system variables
  - `[off] setup:` print the current setup parameters

On/Off

- `[on] statistics:` print the runtime statistics
- `[on] finalstats:` print the final statistics in the end of the execution
- `[on] lowfirst:` sort polynomial expressions by increasing powers
- `[off] highfirst:` sort polynomial expressions by decreasing powers

- These options can be very useful for debugging (when one wants to see as much information as possible and would therefore want to enable the printing out of statistics, setup, etc)
- On the other hand, one can suppress the printing of all statistics so as to get only the final expression printed out [the default FORM greeting message that prints out the version of FORM etc. can be switched off by using the option `-q`: `form -q script.frm`]. This is useful when the FORM output is to be used by another program (e.g., Mathematica). One can get the result in a text file by, e.g., redirecting the output:
  `form -q script.frm > script.log`

[Example file: nomessages.frm]

Checkpointing in FORM

- FORM has a built-in checkpointing mechanism, which is also switched on or off in the same manner:
  - `[off] checkpoint;` activate/disactivate the built-in checkpointing
- We will not consider this here, mainly because FORM is so fast that checkpointing is hardly ever needed; however, it is useful to know that such feature exists, in case you ever run a really very big calculation

Manipulate Functions: Symmetrize

- `symmetrize` (`antisymmetrize`, `(r)cyclesymmetrize`)
  symmetrizes (antisymmetrizes, ...) the arguments of a function or a tensor
  - the symmetrization can be restricted; this is more versatile than defining functions to be (anti)symmetric (which then applies to all of its arguments)
  - `function f;`

    `symmetrize f;` all arguments

    `symmetrize f 2,3,5;` selected args, no action if less than 5 args

    `symmetrize f:4;` only functions with 4 args

    `symmetrize f:4 2,3;` functions with 4 args, selected args

    `symmetrize f: 2,3,5;`
    selected args, action always, args with too large numbers ignored

    `symmetrize f (1,3),(2,4),(7,8);`
    groups of arguments; no action if less than 8 args

    `symmetrize f:6 (1,3),(2,5),(4,6);`
    groups of args, only functions with 6 args

    `symmetrize f: (1,3),(2,4),(5,7);`
    groups of args, action always, groups with too large numbers ignored

Manipulate Functions: Transform

- `transform`
  Transforms arguments of a function/tensor (or functions/tensors) according to specified transformations. [Example file: transform.frm]
- One can specify ranges of arguments to which the transformations are applied, such as `(1,5)`, `(1,last)`, `(last-3,5)`; these specifications can include $-variables provided they evaluate to numbers at runtime
- The syntax is as in the example:
  `transform f, reverse(1,last), dropargs(3,5);`
  this will reverse all arguments of `f`, and then drop arguments 3 to 5 thereof.
- Some useful tranformations include:
  - `reverse(m,n);` reverses args from `m` to `n`
  - `permute(m,n,k)(p,r)…;` permutes args once in the specified cycles
  - `dropargs(m,n);` removes args from `m` to `n`
  - `selectargs(m,n);` removes args outside the range `m` to `n`

Manipulate Functions: Transform

[Example file: transform.frm]

- Some useful tranformations – continued:
  - `addargs(m,n);` replaces args from `m` to `n` with their sum
  - `mulargs(m,n);` replaces args from `m` to `n` with their product
  - `dedup(m,n);` removes duplicate args in the range, keeping the first
  - `replace(m,n)=(rules);` this is a replacement transformation which is applied to the range from `m` to `n` according to the replacement rules. The rules are a list of pairs where the first element shows what to replace, and the second element what to insert, such as `(x,y,a,b,1,2)`. The rules allow simple replacement of generic arguments, which are denoted by the special variables `xarg_`, `iarg_`, `parg_`, and `farg_`, such as `(xarg_,xarg_+2,parg_,2*parg_)`, which will replace scalarlike arguments by their sum with 2, and vectorlike arguments by their product by 2. Note that `iarg_` and `farg_` do not seem to work at the moment! Replacements are done only once, and they also allow mixing of types (e.g., a symbol can be replaced by a vector etc).

Manipulate Functions: Arguments

[Example file: argument.frm]

- `argument;`
  …
  `endargument;`
- As already discussed, this environment allows one to work (i.e., apply identify statements) at the level of function arguments
- Similarly to symmetrization statements, one can restrict the scope of this statement:

`argument;`	applies to all arguments of all functions
`argument 2;`	applies to second arguments of all functions
`argument f;`	applies to function `f`
`argument f,5,7;`	applies to arguments 5 and 7 of function `f`
`argument {f,g,h};`	applies to functions `f, g, h`
`argument setfun,1,3;`	applies to functions that enter set `setfun`, their arguments 1 and 3

- The last four patterns can repeat as many times as needed:
  `argument 2,f,1,{F,G,H},3,4,g,5;` applies to second arguments of all functions, the first argument of `f`, arguments 3 and 4 of `F, G`, and `H`, and argument 5 of `g`

Manipulate Functions: Chain In/Out

- `chainin f;`
  collects products of `f` with different arguments into a single instance of `f` with many arguments, is equivalent to, but much faster than
  ```
  repeat;
  id f(a?)*f(b?) = f(a,b);
  endrepeat;
  ```
- `chainout f;`
  does the opposite to `chainin f`, is equivalent to, but much faster than
  ```
  repeat;
  id f(a?,b?,?c) = f(a) * f(b,?c);
  endrepeat;
  ```

Functions: Commuting Sets

- As we discussed before, functions or tensors can be declared non-commuting and commuting, and the default situation is that non-commuting functions or tensors do not commute with any other non-commuting functions or tensors
- Sometimes this is not what we want; the default behaviour can be changed by the statement `commuteinset`:

```
functions f, g, h, F, G, H;
commuteinset {f,g,h},{F,G,H},{F,F};
```

- Non-commuting functions or tensors specified in a set in the `commuteinset` statement will commute with other functions from the same set
- Note that in order to make a function commute with itself (which is not necessarily the case for non-commuting functions of different arguments) it has to be specified twice within the same set. Note also that a function can appear in more than one set

Manipulate Expressions and Terms

- Here we will consider the details of some statements that allow one to manipulate expressions either as a whole or on a term-by-term basis. Some of these statements have already been encountered
  - `Global expr = something;` defines a global expression
    `Global expr;` makes existing local expression `expr` global
  - `Local expr = something;` defines a local expression
    `Local expr;` makes existing global expression `expr` local
  - `drop expr1, expr2, ...;` removes expressions from the list from the system (they can still be used in the rhs of statements such as `Global expr3 = expr1`). After the end of the current module these expressions are eliminated completely. If the list of expressions is empty, all previously defined expressions are removed.
  - `ndrop expr1, expr2, ...;` cancels the previous instructions to drop the listed expressions (or all expressions if the list is empty)

Manipulate Expressions and Terms

- `skip expr1, expr2, ...;` skips listed expressions (or all expressions if the list is empty) from the current module. The skipped expressions are not affected by the module statements but can be used in the rhs of assignments.
- `nskip expr1, expr2, ...;` cancels the previous instructions to skip the listed expressions (or all expressions to be skipped if the list is empty)
- `hide expr1, expr2, ...;` hides listed expressions (or all expressions if the list is empty). The hidden expressions are inactive until they are restored by the `unhide` statement, but can be used in the rhs of assignments
- `unhide expr1, expr2, ...;` unhides listed expressions (or all hidden expressions if the list is empty)
- `nhide/nunhide expr1, expr2, ...;` work in analogy to `nskip` and cancel the instructions to `hide/unhide` the listed (or all affected) expressions

Manipulate Expressions and Terms

- `term;`
  `statements;`
  `endterm;`
  This environment allows one to work with separate terms [in expressions] as if they were expressions themselves, i.e., simplify and sort terms on their own. Only executable statements (and the term-by-term `print` statement, to be considered in the next lecture) are allowed inside, and the sorting is done with a special instruction `sort` (not to be confused with the `.sort` end-of-module statement)
- `multiply [left/right] expr;` Multiplies all terms on the left or on the right by the specified expression (either given explicitly or an existing one). The option left/right can be omitted, but the corresponding result, in case of non-commuting objects, can depend on many things, so it is better to specify this option unless there are only commuting objects

Manipulate Expressions and Terms

- `inside $var1, $var2, ...;`
  `statements`
  `endinside;`
  This environment is similar to the `term;...endterm;` statement, and it allows one to work with the listed $-variables separately. Only executable statements are allowed inside. Note that the $-variables are treated one by one, so that their values change such that, e.g., `$var2` uses `$var1`, the new value of `$var1` will be substituted in `$var2`, but if `$var1` uses `$var2`, the latter will be substituted with the old value

- `discard;` This statement discards the current term; it can be useful in, e.g., conditional instructions such as
  `if (count(x,1) > 3) discard;`
  which discards all terms with power of `x` greater than `3`.