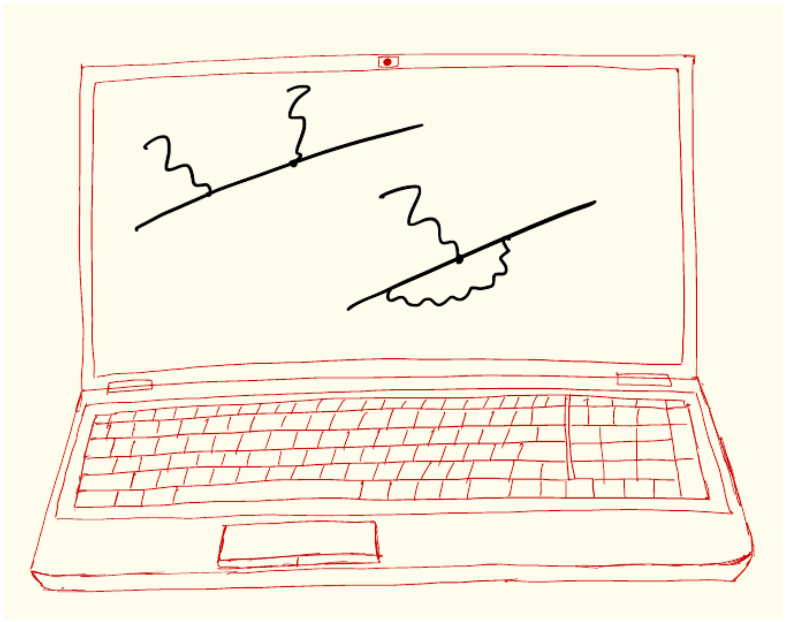
Computer Algebra for Feynman Graphs



4.

In this Lecture

- Some more tricks in pattern matching, working with coefficients, polynomials, and ratios of polynomials, and factoring out terms:
  - PolyFun (PolyRatFun)
  - Bracket/Antibracket
- FORM Preprocessor: variables, calculator, #do cycle

Pattern Matching: Coefficients

- Wildcards don’t match numerical coefficients in polynomials:

```
Local expr = x^2 * y + 2 * y;
id a? * y? = func(a^3 * y^2);
=> expr = func(x^3 * y^2) * x + 2 * y;
```

[Example file: polyfun.frm]

  - the last term didn’t get matched!

- There is a nice way to extend the scope of wildcards to numerical coefficients: PolyFun option to an end-of-module directive. It turns all numerical coefficients (including 1) into arguments of a *selected predefined* function (here named `pfun`); since wildcards match numerical arguments, it solves the issue:

```
Local expr = x^2 * y + 2 * y;
.sort (PolyFun = pfun);            [note the syntax!]
=> expr = pfun(1)*x^2*y + pfun(2)*y;
id pfun(1) = 1;                    [if we don’t want to keep pfun(1)]
id a? * y? = func(a^3 * y^2);
id pfun(a?) * y? = func(a^3 * y^2);
=> expr = func(x^3 * y^2) * x + func(8 * y^2);
```

  - we have the desired match!

PolyFun: Working with Coefficients

[Example file: polyfun.frm]

- `Polyfun` can also be used to define a function in the declarations section, in which case it is assumed to be a (commuting) function of one argument:
  ```
  polyfun pfun;
  Local expr=6*pfun(5*a)*x^2+2*pfun(6*b)*x^2+pfun(x, y)*x^2;
  ```
- If there are two equal terms multiplied with different polyfuns, the arguments of these polyfuns are added up (with the external coefficients pulled in):
  ```
  => expr = x^2*pfun(12*b + 30*a) + x^2*pfun(x,y);
  ```
- If there are more than one arguments – `pfun(x,y)` – it is treated as a regular function; the same happens if the arguments are not scalars
- There can be only one function declared as `polyfun`
- Polyfun declaration can be reverted (in one of the following modules) by using it without arguments; expressions do not change, so if one wants to turn the arguments of polyfun back into coefficients, this needs to be done manually
  ```
  .sort
  polyfun;
  Local expr2 =  2 * x * pfun(z) + 3 * x * pfun(1);
  ```

PolyRatFun: Working with Rationals

- A similar declaration, `PolyRatFun`, exists that allows working with coefficients that are rational fractions:

  [Example file: polyratfun.frm]

  ```
  polyratfun prfun;
  Local expr = 2*x^2*prfun(a,b)+3*x^2*prfun(c,d)+prfun(a);
  => expr =x^2*prfun(2*a*d+3*b*c,b*d)+prfun(a)*prfun(1,1);
  ```
- `PolyRatFun` can have at most two arguments, both being numbers or symbols. The first is the numerator, the second is the denominator, so that `prfun(a,b)` means `a/b`; numerical coefficients are pulled inside
- The coefficients of equal terms are added up
- Instances of PolyRatFun with one argument are treated as generic functions, whereas having more than two arguments generates an error message
- One can also define a second `PolyRatFun` being the inverse of the first:

  ```
  polyratfun ratio, inverse;
  ```
  FORM knows that `ratio(a,b) = inverse(b,a)`, so that
  ```
  Local expr = 2 * x^2 * ratio(a,b) + x^2 * inverse(b, a);
  ```
  gives
  ```
  expr = x^2 * ratio(3*a,b)
  ```

PolyRatFun: Expanding & Divergences

- `PolyRatFun` can be used to keep track of (rational) divergences and to expand polynomials [Example file: polyratfun.frm]
- `PolyRatFun fun(divergence, x)` keeps only the most divergent piece in `x` (with the coefficient 1), so that one knows what is the leading divergence
- `PolyRatFun fun(expand,x,n)` expands the ratio in powers of `x`, up to `(n-m)`th power, where `m` is the power of the leading divergence. In this usage, the denominator of `PolyRatFun` can only be a polynomial in `x`
- After the expansion, the remaining instances of `fun` will have only one argument and will behave as `PolyFun`, i.e., equal terms will have their coefficients (arguments of `fun`) summed inside of the common factor of `fun`
- This way of using `PolyRatFun` is very useful when keeping track of divergences (e.g., in the dimensional regularisation)
- The above behaviour is very efficient: instead of keeping track of the complicated rational expressions, FORM just keeps only the necessary terms

More Factoring Tools: Brackets

[Example file: brackets.frm]

- These two statements provide more tools for manipulating coefficients (and are also useful in output control):
  `bracket a, b, c, …;` objects in the list are factored out and taken outside of the brackets, everything else is kept inside the brackets
  `antibracket a, b, c, …;` objects in the list are kept inside the brackets, everything else is taken outside the brackets
- Objects in the lists can also be sets, which is the same as listing elements of the set separately
- General rules that concern brackets and antibrackets:
  - Only one instance of bracket or antibracket is allowed per module; if there are more than one, only the last has effect
  - Bracketing information is lost in the next module after the declarations (that is, when all brackets are normally expanded by FORM)

Brackets: Manipulation of Elements

[Example file: brackets.frm]

- Expressions that have been bracketed can be addressed separately term by term (in the subsequent module):

```
symbol x, y, n;
Local T = (x + y)^4;
b x, y;
.sort
Local sumc = sum_(n,0,4,T[x^n * y^(4-n)]);
=> sumc = 16 = 2^4
```

- Bracketing information can be used by the `collect` statement (again, in the subsequent module):

```
function f;
Local expr = x^2 * y + 2 * x + 1;
bracket  x;
.sort;
collect f;
```

  - this collects the brackets inside the arguments of `f`, which can then be used in matching similarly to `PolyFun`

Brackets: Keeping Intact

- Apart from collecting the contents of the brackets, they can be kept, i.e., no manipulation is done on them; the pieces outside of the brackets are evaluated and then multiplied by the brackets

```
bracket …;
.sort;
keep brackets;
…
print;
.end
```

- This can be useful sometimes when one does not want the bracketed pieces to be changed, or when they contain very long expressions so that the term-by-term evaluation would be much lengthier than simplifying the coefficients
- Be careful when using this option: for instance, forgetting one of a pair of indices inside the brackets and summing over that index will give a wrong result

FORM Preprocessor

- The preprocessor part of FORM prepares (=edits) the input for the compiler
- The preprocessor has its own variables and commands
- Preprocessor variables have to be defined; their names can be any strings starting with a letter, and they contain strings of characters (uninterpreted)
- When they are used their names have to be enclosed between a backquote and a quote symbols; nesting of these is possible
  `` `a', `b', `c`i'', `x`j'z' ``
- There are predefined preprocessor variables, such as `DATE_` (the current date), `CMODULE_` (the number of the current module), and so on
- There are also predefined macros; two of them exist currently, which are `TOLOWER_(string)` and `TOUPPER_(string)`; they convert their argument to lowercase or uppercase, respectively, and pass the result on as input
- Preprocessor commands start with the hash symbol:

```
#define i "2";
id x = `i';
```

FORM Preprocessor: Calculator

- FORM preprocessor can evaluate simple expressions that involve preprocessor variables; in order to have that, the expressions have to be included in curly braces, and the string (including the braces) is replaced by the result of the evaluation:
```
#define i “5”;
Local T = F({`i’+1}) – F({`i’-1});
=> T = F(6) – F(4);
```
- Valid symbols in arithmetic expressions are the decimal digits and
`+ - * / % ( ) { } & | ^ !`
- The arithmetic operators are as usual, `with + - * /` being obvious, and a few more involved being the following: `%` remainder, `^` exponent, `!` factorial, `&` bitwise and, `|` bitwise or, `^%` postfix base 2 log, and `^/` postfix square root
- The preprocessor does only integer arithmetics, so `{(`i’-1)/(`i’+1)}=0`
- If the preprocessor does not determine the content of a pair of curly braces to be a valid arithmetic expression, the whole string (including the braces) is left as it is

FORM Preprocessor: Calculator

- The comma is not a valid symbol in the preprocessor arithmetic expressions
- This allows one to avoid issues when implicitly declaring numerical sets – recall the example from the previous lecture:

  `id x^n?!{-1} = x^(n+1)/(n+1);` the preprocessor will turn that into

  `id x^n?! - 1 = x^(n+1)/(n+1);` which is invalid syntax
- This issue is avoided by putting in a comma:

  `id x^n?!{-1,} = x^(n+1)/(n+1);` this is not processed by the preprocessor

FORM Preprocessor: Repeat Patterns

[Example file: preprocessor.frm]

- Definitions with repeated patterns can be made easier with the use of the preprocessor triple dot operator ...
- The simplest use of this operator is as in the sequences here:
```
Local T = a1 +...- a12;
Local F = b0 +...+ b22;
Local G = f(b0,...,b27);
Local H = a1 * ... * a10;
Local J = 1/b1/.../b6;
```
- More general repeated patterns follow this scheme:
`<pattern1>operator1...operator2<pattern2>`
- The pair of operators can be one of these: (++), (--),(+-),(-+),(**),(//),(,,)
- The two patterns can be more complicated; their difference can be only in numerical characters (the counters)
- They can have more than one counter, the counters can run in different directions, the increment of each counter, however, is always +1 or -1 (or 0 if they initially are the same, i.e., equal numbers in the patterns do not cause action)

FORM Preprocessor: Repeat Patterns

[Example file: preprocessor.frm]

- More general repeated patterns follow this scheme:
  `<pattern1>operator1...operator2<pattern2>`
- The pair of operators can be one of these: (++), (--),(+-),(-+),(**),(//),(,,)
- Example:

```
Local complicated = <fa1b2c15f(g6)>+...-<fa5b6c11f(g2)>;
```

- Note that the lengths of ranges of all counters should be the same (syncronised), otherwise there is an ambiguity as to whether some ranges have to be extended or others to be shrunk, and an error message is generated

FORM Preprocessor: #do … #enddo

- The preprocessor incorporates means for the program flow control, the most useful of which is perhaps the do loop:

```
#do ivar = istart, iend, increment
… something to do;
#enddo
```

[Example file: do_cycle.frm]

- Here, `istart`, `iend`, and `increment` are integers that are taken, respectively, as the starting value of `ivar`, the final value at exceeding which the cycle ends, and the increment (the latter value can be omitted, with the default increment +1).
- Since `ivar` is a preprocessor variable, its value has to be referred to as `` `ivar' `` in the body of the cycle
- There are the following variations of the do cycle:
  - Listed do: `#do ivar = {string1,…, stringN}` evaluates for `ivar` equal to each of the listed strings;
  - Expression do: `#do ivar = expr` evaluates for `ivar` equal to each of the terms in `expr`, term by term. A copy of `expr` is stored to protect from changes of terms during the evaluation of the cycle