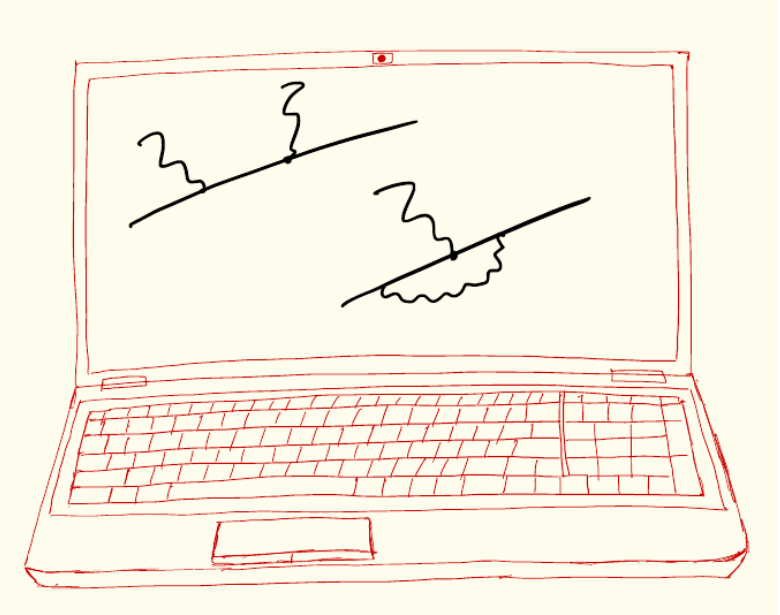
Computer Algebra for Feynman Graphs



8.

In this Lecture

- Input control: #include, folds, save/load
- Output control: printing, output formats, dictionaries, ...

Input: #include

- A simple way to make FORM definitions and statements reusable over and over again is to write them into a text file (which has to comply with the usual FORM syntax). Let us call this file `statements.h`.
- This file can be included in the text of a FORM program (which is done at the level of the preprocessor) by using the following directive:
  `#include statements.h`
- The preprocessor reads the text of statements.h and appends it to the text input at this point. The `#include` directive can be followed by a switch (`+` or `-` sign) that works as a `#+` or a `#-` instruction w.r.t. the contents of `statements.h`.
- If FORM cannot find a file `statement.h`, there will be an error. Be careful: FORM can search for files in many directories (depending on the call options and the value of the `FORMPATH` variable), so make sure FORM actually finds what you want it to!

[Example file: include.frm]

Input: #include with Folds

- The contents of the included file can be structured in sections, called folds, that are declared as follows (this has to do with code folding):

```
*--#[ foldname :
...
statements;
...
*--#] foldname :
```

Please note the syntax: it is quite rigid here, in particular, there has to be no spaces before the *'s (which in fact are the current commentary character)

[Example file: include.frm]

- Each of these folds can be included separately by the `#include` directive:
  `#include statements.h #foldname1 #foldname2 ...`
- This makes the preprocessor read the contents of the fold with the specified name and append it to input. The rest of the included file is ignored.
- Using folds enables one to include blocks of statements in a more flexible way, which can be used as another means of "procedural" programming (along with preprocessor variables/macros and preprocessor procedures)
- Since #include works at the level of the preprocessor, one can even have parts of the input (rather than complete statements) in the included file

Expressions: Save/Load

- Apart from storing definitions such as blocks of statements (and other pieces of input) and possibly expressions in text files and inserting them in the input using the #include directive, one can store, save and load complete expressions (e.g., when the result of a calculation in FORM needs to be further processed in another program in FORM). This is done using the instructions `.store, save, and load`:
- `.store` stores all active global expressions
- `delete storage` empties the storage file
- `save filename.sav [expr1, expr2, …]` saves the listed (or all if the list is empty) stored expressions to the file `filename.sav` (note the conventional, although not obligatory, filename extension)
- `load filename.sav [expr1, expr2, …]` loads the listed (or all if the list is empty) expressions from the file `filename.sav` into storage. The loaded expressions become stored global expressions; an error occurs if the name of an expression was listed but not found in the save file

[Example file: saveload.frm]

Save/Load: Stored Declarations

- Note that apart from the stored expressions per se, FORM also stores information about the objects that occur in those expressions, i.e., the declarations of symbols, vectors, indices etc.
- This means that these objects become known to FORM when the save file is loaded (as can be checked using the `On Names` instruction)
- Conflicting definitions for objects (those changing types, e.g., if `z` is defined in the code as a vector, the code then loads an expression that contains a symbol `z`, and this expression is used in assignment) will result in errors

[Example file: saveload.frm]

Output: Printing

- We have already encountered the `print` statement, which exists in two forms: it either prints expressions in the end of the current module, or prints separate terms during evaluation

[Example file: print.frm]

- The first form:
  `print [expr1, expr2, …]` assigns listed expressions (or all expressions if the list is empty) to be printed at the end of the current module
  `nprint [expr1, expr2, …]` removes the listed (or all) expressions at from the list of expressions to be printed at the end of the current module
- The second form:
  `print "format string" [objects]` prints separate terms or objects such as $-variables during evaluation; it can be encountered in different parts of the program. The useful format control characters are:
  `%t (%T)`: the current term with (without) the possible leading plus sign;
  `%$`: a $-variable, the name of the $-variable to be printed follows the format string as in `"printout: %$" $a`
  `%:` prevent newline after the printout (having one is the default)
  `%%`: print a % character; `\n` print a newline character

Printing Options

- A print command can be followed by options such as

  `print +f -s;`

  `+f/-f` switch off/on output to the screen; this works only if output to a log file is used by specifying a `-l` option in the call of FORM, the default is `-f` (output both to the screen and the log file). This option can be used in both the full expression mode and term-by-term mode of the print statement

  `-s/+s/+ss/+sss` switch modes of printing terms, with `-s` (default) printing as many terms in one line as possible within the specified length, `+s` printing each term on a single line, `+ss` printing each group (a function, all symbols together, all vectors together, all dotproducts together) on a single line, and `+sss` printing each object on a single line

[Example file: print.frm]

Output: Formatting

- FORM allows to control in what form the output is printed out, using the format statement:
  `format [options]`
  [Example file: format.frm]
- Options include:
  `fortran/doublefortran/quadruplefortran/fortran90/C/mathematica/maple`: format output in a form readable in the corresponding programming language (may not work perfectly, generally works fine)
  `nospaces/spaces`: remove/put spaces (the latter is the default)
  `39<N<255`: a number, the requested length of line; the default is `72`
  `float <N>/rational`: switches on printing numbers as floating point/rational (the latter is default). `N` is the floating point precision (the default is `10`)
- Format statements may follow each other; they are either combined (e.g., `format C; format 120`) or the last format overrides the previous ones (e.g., `format fortran90; format mathematica`)

Output: Dictionaries

- FORM also allows for a finer control over the output formatting, using the mechanism of user-defined dictionaries; this can be used to adjust output to the specific needs of post-processing (e.g., LaTeX, Fortran, etc)
- In a dictionary, one can define words and their “meanings” with which they will be replaced in the output
- A dictionary has to be opened in order to add entries:
  `#opendictionary name`
  One can define many dictionaries, but only one can be open at any instance.
- Entries are added to a dictionary as follows:

```
#add x1: "x_1"
#add 2:   "two"
#add *: "\,"
#add ^: "**"
```

Allowed in the l.h.s.: variables (a symbol/index/function), numbers (integer/rational), special characters (*,^), a function with arguments

[Example file: dictionary.frm]

- Strings in the r.h.s. can be anything (but keep in mind how the preprocessor works, e.g., [LaTeX] braces might be considered as calculable expressions and evaluated unless a non-calculable character is inserted)

```
#add 1/2: "\frac{1,}{2,}"
```

Output: Dictionaries

- After entries in the dictionary have been added, it has to be closed:
  `#closedictionary name`
  One can define many dictionaries, but only one can be open at any instance.
- A dictionary can be used in the output by invoking the directive
  `#usedictionary name (option1,option2,...)`
  where options regulate how the dictionary is used, i.e., what type of objects is looked up in the dictionary

  [Example file: dictionary.frm]
- (Some of) the options are:
  - `allnumbers`: all numbers are looked up in the dictionary
  - `integersonly`: only integers are looked up
  - `nonumbers`: numbers are not looked up
  - `numbersonly`: numbers are not looked up
  - `novariables`: vars (not numbers/special characters) are not looked up
  - `variablesonly`: only variables are looked up

Output: Dictionaries

- (Some of) the options, continued: [Example file: dictionary.frm]
  - `nospecials:` special characters (*,^) are not looked up
  - `specialsonly:` only special characters are looked up
  - `nofunwithargs:` functions with arguments are not looked up
  - `funwithargsonly:` only functions with arguments are looked up
  - `warnings:` floating point warnings in fortran or C formats, warning if the dictionary cannot be used so as to avoid floating point notation
  - `nowarnings:` no floating point warnings
  - `infunctions:` dictionary is used inside function args
  - `notinfunctions:` dictionary is not used inside function args
  - `$:` substitutions are made in $-variables
- The default is that all possible matches are looked up in the dictionary, but no warnings are given and no $-variables are expanded