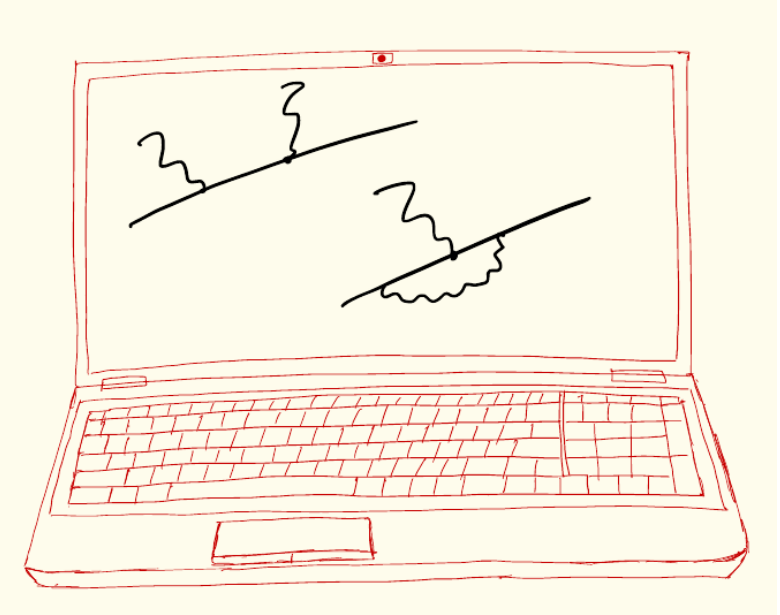
Computer Algebra for Feynman Graphs



1.

The Basics of FORM

- Recipe: how to use FORM

1. Download the executable from the FORM website (the sources are also available in gitHub, so you can even compile it on your own machine)
2. Change the necessary permissions
3. Make and edit your first script
   - script: a text file, can be edited in any* text editor
   - **has to have** a special extension: `*.frm`
4. Run FORM feeding your script to it: `form your_script.frm`
5. See what you get and decide if you are happy with the results
6. Repeat steps 3-5 if needed

- Important: be careful what you ask for!

In this Lecture

- Basic syntax
- Expressions and objects [~variables] in FORM
- Types and declarations of variables (including automated [implicit] declaration)
  - Functions
  - Indices and vectors (and the SCHOONSCHIP notation)
  - Tensors
- [Some] intrinsic objects (selected mathematical functions, metric tensor, Levi-Civitta symbol, γ matrices, …) with examples
- **Substitutions** (`identify` statement)

“Hello, world!” in FORM

- Let’s write a very simple script: [Example file: Basic_1.frm]

```
Symbols a, b, c, x, z;

* ← This is the default comment symbol

Local Helloworld = (a + b + c + x + z)^3;
Local [Hello, world!] = (a + b)^2;

print;
.end
```

- “`Symbols`” with a list defines a list of, well, “symbols” - objects that behave like “normal” numbers, for instance, they commute with every other objects
- “`Local`” defines our main expression(s) that is(are) being operated on
- “`print`” tells FORM to output (either all expressions or those specified after the command)

Syntax Conventions

- FORM code is built of statements, all of which look like this:

```
keyword expression;
```

- Statements:
  - more than one in line possible
  - a single statement can take more than one line
  - end with a semicolon`;`
- **Keywords** (such as `symbol`, `Local`, and `print` in the example above) are **case-insensitive**: `Symbol`, `SyMBoL`, `SYMBOL` are all the same
- The same is true for other built-in objects
- Keywords can be shortened (I personally try to avoid it for the sake of clarity): `s a` is the same as `symbol a`, `p` or `pri` is the same as `print`. [Check it!]
- Moreover, `symbol` and `symbols` are the same (as are similar declarations)

Syntax Conventions

- User-defined objects are case-sensitive: `symbol MySymbol` and `symbol mySymbol` declare two different objects!
- Object names can be any sequences of letters and digits, starting with a letter (very intuitive), such as

```
symbol a, b, a3431, theansweris42;
```

- More sophisticated names (containing special symbols) are possible if one uses the square brackets:

```
symbol [Hey, look at this symbol! It's so funny!];
```

Remark: the preprocessor appears to strip spaces/replace them by commas (seems to be undocumented), beware!

- The underscore symbol is a bit special: it appears in many built-in symbols, so it is a good idea to avoid using it at all (explicitly forbidden in newer versions):

```
symbol mysymbolwithanunderscore_;

symbol d_;
```

Hmm… Maybe you can get away with this… or not

Bad idea: `d_` is an intrinsic symbol! (Will talk about them later)

Useful Commands, Pt. 1

- Switch off/on the printing out of output:

  `#-` and `#+`
- Switch off/on runtime statistics (how many terms are generated etc):

  `Off/On statistics`
- Switch off/on listing of defined objects (useful when debugging)

  `Off/On names`
- Let us see how they work: try inserting them in the example above!

Functions

- Let us introduce more sophisticated objects – functions:

[Example file: Functions_1.frm]

```
Functions F, G, H;
CFunctions f, g, h;
Commuting phi, rho, eta;
```

- They can have arguments (not necessarily):

```
Local expr=(F(x) + G(x))^2;
Local cexpr=(f(x) + g(x))^2;
Local args=f(x) + f + g(x,y) + h() + f(x, ,y);
```

- Empty argument = zero argument [not always, note the behaviour of h( )]
- Functions do not commute with other functions; CFunctions (=Commuting) do commute with other CFunctions and with functions
- FORM does not watch after the types of functions and the number of variable in each function until it is asked to do something (maybe incompatible with the number of arguments)

Functions: Symmetries

- Functions can have definite symmetry under permutation of arguments:

[Example file: Functions_2.frm]

```
Functions A, B, C, D, F(symmetric), G(antisymmetric),
H(cyclic), J(rcyclic);
Local symexpr=(F(x,y,z) + F(x,z,y))^2;
Local antisymexpr=G(x,y,z) + G(x,z,y) + G(z,x,y);
Local cyclicexpr=H(x,y,z)+H(z,x,y)+H(x,z,y)+H(y,x,z);
```

- Symmetry properties are applied automatically (and keywords can be abbrev.)
- A function can have symmetry only with respect to the whole set of its arguments; if only a subset needs to be symmetric (or different subsets have different symmetries), a product of functions, one for each subset of arguments, will have to be used
- Symmetries can be applied to general functions (i.e. those that were not defined as symmetric), using the statements

```
[anti,cycle,rcycle]symmetrize A;
```

Functions: User-defined and Intrinsic

- User-defined functions allow for evaluation of more sophisticated expressions (e.g., those with non-commuting elements and with nested dependence)
- A user-defined function without an argument is essentially a (non-commuting) symbol
- Apart from user-defined functions, there are a few implemented intrinsic mathematical functions, some of which are given below:
  - `abs_(x)=|x|`
  - `binom_(k,n)=n!/k!/(n-k)!`
  - `max_(x,y,z,…)`
  - `fac_(n)=n!`
  - `invfac_(n)=1/n!`
  - `min_(x,y,z,…)`

  [see manual for other implemented functions]; they are evaluated only when their arguments are (all) numerical (and integer for factorials etc).
- Excercise 1. Investigate the behaviour of intrinsic functions.
  Write a short FORM script which would call the different intrinsic functions with different (numerical [integer/non-integer], non-numerical [symbolic]) arguments, and see what output you get.

Vectors and Indices

- Two more data types, natural for (particle) physics: vectors and their indices
  [Example file: Indices_1.frm]

```
vectors p, q, r, s;
indices m, n, a, b;
```

- The default dimension of vector indices is 4; can be changed explicitly:

```
dimension 3;
index al=2, be=4, l=0;
```

- If there are two equal indices, the Einstein summation is applied:

```
Local dotpq = p(m) * q(m);
```

- Note: FORM does not complain if there are **more than two equal indices**; it just sums over the first two repeating indices that it encounters, and so on, and the order can depend on many things, so it is up to the user to **avoid** that!
- The Einstein summation is not done if the dimension of the repeating indices is zero; the summation then can be done explicitly, if needed [considered below]

Vectors and Indices

- FORM does not distinguish between covariant and contravariant vectors or indices; these can be implemented using a metric tensor [which is usually not needed in particle physics but might be useful in gravitation etc.]
- Vectors are commuting objects
- Vectors can only appear with indices (predefined symbolic or numeric):
  `p(m), q(1)`
- Dot products can be written as

  ```
  p.q = p(m) * q(m);
  ```

- The SCHOONSCHIP notation: if a function depends on an index, and a vector has the same index, the result of the Einstein summation is written as
  `f(m,n) * p(m) * q(n) = f(p,q);`
- The statement
  `sum k, m, l, …;`
  forces a summation over the specified indices

Tensors

- Tensors (Ntensors) are special (non-)commuting functions that can only depend on indices and vectors: [Example file: Tensors_1.frm]

```
Tensors t, s;
Ntensors w, g;
```

- The SCHOONSCHIP notation applies to tensors, too

```
Local tuv = t(u,v);
Local S1 =
          w(m,n) * u(m) * v(n) * g(p,q) + g(p,q) * w(u,v);
Local S2 = t(m,n)*s(n,a) + t(m,r)*s(r,a);
```

- In the last expression, one needs to tell FORM explicitly that the two terms are the same, this can be done by forcing summation over `n` and `r`

```
sum n, r;
```

- One needs to be careful: FORM replaces the indices that are summed over with dummy indices; any hanging indices will turn into dummy indices too: `t(m,n)` → `t(m,N1_?)`

Intrinsic Tensors

- Some of the useful intrinsic tensors in FORM are:
  - Kronecker delta symbol (metric tensor) `d_(i,j)`
  - Levi-Civitta symbol `e_(i,j,k,...)`
  - Dirac matrices `g_(i)`
- Note that one does not have to use them – one can define the corresponding objects explicitly [and thus be independent of the intrinsic definitions]
- The Levi-Civitta symbol is perhaps the trickiest – that is one of the places where the difference between the covariant and contravariant components matters; recall that in the Minkowski space

$$(1/4!)\epsilon^{ijkl}\epsilon_{ijkl} = (1/4!)\epsilon^{ijkl}\epsilon^{abcd}g_{ia}g_{jb}g_{kc}g_{ld} = \det g = -1,$$

  which implies a relative minus sign between the covariant and contravariant components of `e_(i,j,k,l)`. This sign is not there in FORM!

Levi-Civitta Symbol

- Example: determinant identities with the Levi-Civitta Symbol in D=3:

$$\epsilon^{ijk}\epsilon^{lmn} = \det \begin{vmatrix} \delta_{il} & \delta_{im} & \delta_{in} \\ \delta_{jl} & \delta_{jm} & \delta_{jn} \\ \delta_{kl} & \delta_{km} & \delta_{kn} \end{vmatrix}$$

and so on

- Note the `contract` statement that actually implements these identities:

[Example file: LC_1.frm]

```
dimension 3;
indices i, j, k, l, m, n;
Local d3=e_(i,j,k) * e_(l,m,n);
Local d2=e_(i,j,k) * e_(i,m,n);
Local d1=e_(i,j,k) * e_(i,j,n);
Local d0=e_(i,j,k) * e_(i,j,k);
contract;
print +s;
.end
```

- Output control: `print +s` outputs each term in a new line

Implicit Declaration of Objects

- In FORM, all objects that are used in statements have to be declared:
```
vectors p, q, u, v;
indices m, n;
```
- There is a statement autodeclare that makes it implicit:
```
autodeclare indices mm, nn;
Local L1 = u(mmx) * v(mmx) * p(nny) * q(nny);
```
- Automatic declaration can be overriden by explicit declaration:
```
autodeclare vectors p, q;
index p1, q1;
Local L2 = p2(p1) * q2(p1) * p3(q1) * q3(q1);
```
- Generally, it is a good idea to avoid this, because it can easily lead to errors!
- Automatic declaration, on the other hand, can be very useful, especially if you have to define many objects
- One can also declare sequences:
```
vectors u1,…,u10;
```

Working the Expressions

- We can now define many types of useful objects (and already do some things with them such as sum and contract)
- We want to be able to do more, after all, we want to do algebra=apply identities, discard some pieces, and so on
- The one statement to rule them all (well, maybe with some exceptions):
  `identify something = [something else];`
- It matches the pattern on the left-hand side and replaces it with the right hand side; [Example file: Identify_1.frm]
- It does so only once to a single term in an expression, term by term
- Consequently, it can be applied to monomial terms only:
  `id x * a * b * z = w; id b=b^2;` both work fine
  `id x + y = z;` results in an error
- One can apply identities (and do other things) to an expression repeatedly:

```
repeat;
id b=b^2;
endrepeat;
```