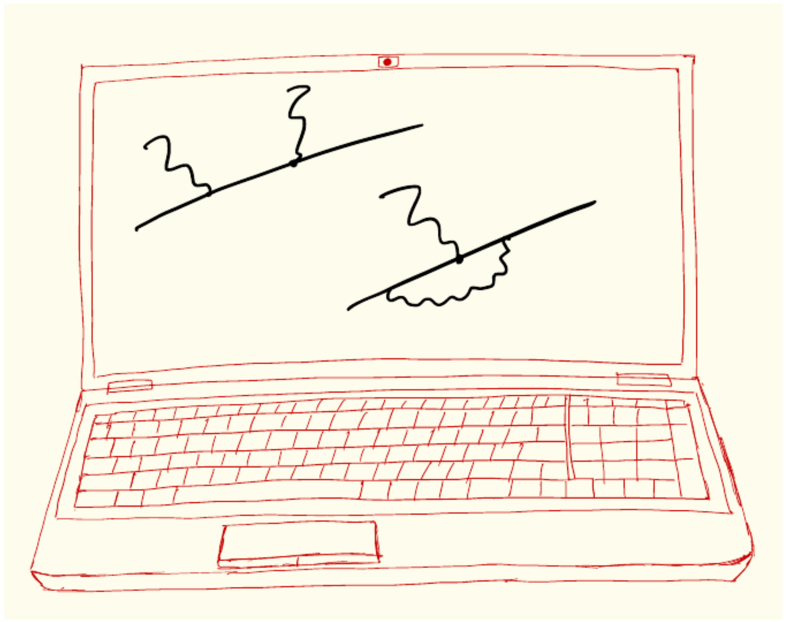
Computer Algebra for Feynman Graphs



3.

In this Lecture

- Matching and replacement control
  - Sets
  - Options in the `identify` statement
- Example: electron Compton scattering

Pattern Matching: Control

- Regular wildcards' scope can often be too broad, for instance, `x?` (with `x` being a symbol) just matches any symbol `x`. We sometimes want to limit this match to only some of the defined symbols, or to exclude some of those from replacement
- FORM allows one to control matching and substitutions using sets and restricting options in the `identify` statement:
  - Sets specify groups of objects that have to be included in or excluded from the matched patterns
  - Options in the `identify` statement tell FORM how exactly it should treat matches should they occur

Pattern Matching: Sets

- A set is a defined group of a few defined objects (symbols, vectors, …):

  [Example file: sets.frm]

  ```
  symbols a, b, c, x, y, z;
  functions f, g, h;
  set sab: a, b, 1;
  set ff: f;
  ```
- All objects in a set have the same type; the first object in a set defines the type. Sets can also contain numerical elements.
- The name of a set should be unique – different from any other names used in a program
- Sets are used in order to have more control over the substitutions: `x?sab` means "any symbol from set `sab`", so that the statement `id x?sab = c` matches and replaces only symbols that enter `sab` and leaves other symbols untouched, and so on
- Sets can also be excluded by adding an exclamation mark "`!`": `f?!ff` means "any function except those from set `ff`"; `id f?!ff(x) = g` matches any function (except those from `ff`) of `x`

Sets as Arrays

- Elements of a set can be addressed by its number (starting from 1):

  [Example file: sets.frm]

  ```
  set syma: a, b, c, n;
  set symx: x, y, z;
  Local expr = syma[1] + syma[3];
  id x?syma[n] = symx[n] + f(syma[n]) - n;
  ```

  Here, `n` is automatically defined as a symbol wildcard (note that symbol `n` still has to be defined), and has the meaning of the position of the matched element `x` of set `syma`, so that results in

  ```
  expr = x + f(a) + z + f(c) - 4;
  ```

- Elements can be also changed between sets, using the following syntax:

  ```
  id x?syma?symx = f(x);
  ```

  This has the meaning that any match of a symbol from set `syma` has to be replaced with the respective element of set `symx`, as instructed in the r.h.s., i.e., `a` is replaced with `f(x)`, `b` with `f(y)`, and so on, so that

  ```
  expr2 = a + c + g(c);
  id x?syma?symx = f(x);
  ```

  results in

  ```
  expr2 = f(x) + f(z) + g(c);
  ```

Implicitly Declared and Intrinsic Sets

- Sets do not need to be predefined: they can be defined in a wildcard, which is sometimes very useful, and is done using the following syntax:
  `id x?{a,b,c} = f(x)` defines the set `{a,b,c}` with elements to match
  `id x?!{x,y,z} = f(x)` excludes the set `{x,y,z}` from matching
- Elements can be addressed by index, as in declared sets:
  `id x?{a,b,c}[n] = n;`
- Note: a special case of a set containing a single numerical element, such as in
  `id x^n?!{,-1} = x^(n+1)/(n+1);`
  The preprocessor, which will be discussed later in more detail, has to distinguish sets from numerical expressions which have to be evaluated. In this example, the comma in front of `-1` (it could also be put after) tells the preprocessor that this is not a numerical expression (since commas are not allowed), and it leaves this untouched, to be identified as a set by the compiler
- Apart from user-defined sets, there are also built-in sets in FORM, some of which are infinite, such as `int_`, which is a set of symbols (recall that the sets have a type), more specifically, of all integers (limited by the FORM word size)

Intrinsic Sets

- Intrinsic sets in FORM are the following:

`int_`	symbols: all integer numbers
`pos_`	symbols: all positive integer numbers
`pos0_`	symbols: all non-negative integer numbers
`neg_`	symbols: all negative integer numbers
`neg0_`	symbols: all non-positive integer numbers
`symbol_`	symbols: all defined symbols (excludes numbers)
`fixed_`	indices: all fixed indices (such as 1 in p(1) where p is a vector)
`index_`	indices: all indices (fixed and defined)
`vector_`	vectors: all defined vectors
`number_`	symbols: all rational numbers
`even_`	symbols: all even integer numbers
`odd_`	symbols: all odd integer numbers
`dummyindices_`	indices: all indices that were obtained automatically as a result of a summation

- Apart from these, intervals (range sets) can be specified, such as `{>=3,<6}, {>=10}, {<-3},` and so on

Intrinsic Sets in Wildcards

- Intrinsic sets can be used in wildcards in the same way as user-defined sets, which can be very useful: [Example file: sets.frm]

```
symbol x, y, n;
Local expr = sum_(n, - 5, 3, x^n/fac_(abs_(n)));
id x^n?neg_ = 0;
print;
.sort
id x^n?odd_ = 0;
print;
.end
```

Restrictions in Substitutions

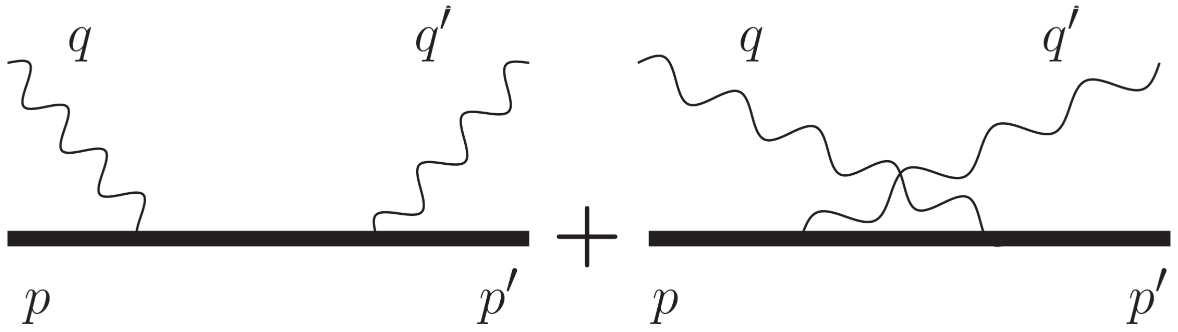
- The default behaviour of `identify` is to try replacing as many matches as possible. This can be altered by adding options to the `identify` statement:
  ```
  id [option] pattern = something;
  ```
- The following restrictive options are available:
  ```
  select <set>
  once
  only
  ifmatch ->   <label>
  ifnomatch -> <label>
  disorder
  ```
- They do the substitutions as follows:
  `select set` performs a substitution only when no elements of the specified `set` are left in a term:
  ```
  set bc: b,c;
  Local expr = a*b*c;
  id select bc a*b = b^2;
  id select bc a*b*c = b^2*c^2;
  expr=b^2*c^2;
  ```

Restrictions in Substitutions

- `once` substitutes the first match that it encounters, all subsequent matches are ignored. Which match occurs the first can be implementation-dependent!
- `only` performs the substitution only if the exact match occurs (i.e., the exact powers of all symbols and dotproducts present)
- `ifmatch -> <label>` skips to the specified label in the program after the substitution, if a match occurs
- `ifnomatch -> <label>` skips to the specified label in the program if no match occurs
- `disorder` works for products non-commuting functions/tensors; a match occurs if the objects in the current term are not in the default order that FORM would assume for them in case they were commuting. Note that the default order depends on the order in which the objects were declared.

Example: electron Compton scattering

- Leading-order (tree-level) electron compton scattering



$$M = e^2 \epsilon_\nu \epsilon'^*_\mu \bar{u}(p') \left[\frac{\gamma^\nu (\not{p} + \not{q} + M)\gamma^\mu}{(p+q)^2 - M^2} + \frac{\gamma^\mu (\not{p} - \not{q}' + M)\gamma^\nu}{(p-q')^2 - M^2} \right] u(p)$$

- We had this as an example earlier; now let us write a code that calculates the square of the amplitude averaged and summed over the helicities of the initial and final particles:

$$\overline{|M|^2} = \frac{1}{4} \sum_{\lambda_i, \lambda_f} \left| M_{\lambda_i, \lambda_f} \right|^2$$

Compton Scattering, Module 1

```
vectors p, pp, q, qp, k, e, ep, ps, pu;
symbols s, t, u, II, Q, M, w, wp;
indices mu, nu, rh, la;
ntensor g;
tensor dd(s);
.global;

Global Ampl(e,ep) =
 (- II * Q * g(ep)) * ( - II * (g(ps) + M)) * (- II * Q * g(e )) / (  2
   * p.q )
+(- II * Q * g(e )) * ( - II * (g(pu) + M)) * (- II * Q * g(ep)) / ( -2
   * p.qp);

id II^2 = -1;

repeat;
id g(?a) * g(?b) = g(?a,?b);
endrepeat;

b g;
format 100;
print;
.store;
```

Declarations

Saving declarations for later

This is the definition of our amplitude

And some simplifications

`Ampl(e,ep)` is now stored and not active; we will use it to define other quantities

Compton Scattering, Modules 2-4

```
Global AmplC(e,ep) = Ampl(e,ep);
id II = -II;
transform, g, reverse(1,last);
.store;
```

Defining Hermitian conjugate amplitude

This reverses arguments of function `g`
Note that it acts only on `AmplC`!

```
Global Square = Ampl(e,ep) * (g(p) + M) * AmplC(e,ep) * (g(pp) + M) /4;

id II^2 = -1;
repeat;
id g(?a) * g(?b) = g(?a,?b);
endrepeat;
.sort
```

Now we define the amplitude squared (inserting the nucleon polarisation sums)

$$\sum_\lambda u_\lambda(p)\bar{u}_\lambda(p) = \not{p} + M$$

The traces of Dirac matrices are implicitly assumed here; we will take care of them later!

```
id ps = p + q;
id pu = p - qp;
```

Here we insert some kinematic identities

```
id g(?a,e,?b,e,?c) = g(?a,mu,?b,mu,?c);
id g(?a,ep,?b,ep,?c) = g(?a,nu,?b,nu,?c);

.sort;
```

And this is the identity to sum over the polarisations of the final and initial photon

Compton Scattering, Module 5

```
repeat;
id g(?a,mu?,mu?,?b) = 4 * g(?a,?b);
endrepeat;

repeat;
id g(?a,mu?,mu?,?b) = 4 * g(?a,?b);
id g(?a,p?,p?,?b) = p.p * g(?a,?b);

id g(?a,mu,nu?,?b,mu,?c)=2*dd(mu,nu)*g(?a,?b,mu,?c)-g(?a,nu,mu,?b,mu,?c);
id g(?a,nu,mu?!{mu},?b,nu,?c)=2*dd(mu,nu)*g(?a,?b,nu,?c)
                  -g(?a,mu,nu,?b,nu,?c);

id g(?a,nu,p?,?b) = 2 * p(nu) * g(?a,?b) - g(?a,p,nu,?b);
id g(?a,mu,p?,?b) = 2 * p(mu) * g(?a,?b) - g(?a,p,mu,?b);

id g(?a,mu?,?b) * dd(mu?,nu?) = g(?a,nu,?b);
id dd(mu?,mu?) = 4;
id dd(p?,pp?) = p.pp;
endrepeat;
.sort;
```

Here we do the main simplifications

Using the main commutation identity for the Dirac matrices, we contract the photon polarisation indices

$$\{\gamma^\mu, \gamma^\nu\} = 2g^{\mu\nu}$$

Note the syntax here!

Compton Scattering, Module 6

```
id g = 4;
id g(p?) = 0;
id g(p?,pp?) = 4 * dd(p,pp);
id g(p?,pp?,q?) = 0;
id g(p?,pp?,q?,qp?) = 4 * (
       dd(p,pp) * dd( q,qp)
      -dd(p,q ) * dd(pp,qp)
      +dd(p,qp) * dd(pp,q )
);

id dd(p?,pp?) = p.pp;
id pp = p + q - qp;

id p.p = M^2;
id q.q = 0;
id qp.qp = 0;
id p.pp = M^2 + q.qp;
id pp.qp = p.q;
id q.qp = p.q - p.qp;

.sort
```

Here we do final simplifications

Recall that we calculate traces of the products of Dirac matrices:

$$\mathrm{tr}\,\mathbf{1} = 4$$
$$\mathrm{tr}\,\gamma^{\mu} = 0$$
$$\mathrm{tr}\,\gamma^{\mu}\gamma^{\nu} = 4g^{\mu\nu}$$
$$\ldots$$

Here we have some final contracttions and kinematic identities

Exercise

- Check that the Compton scattering amplitude is transverse:

$$T = \epsilon'^*_\mu \epsilon_\nu M^{\mu\nu} \qquad \text{where} \qquad q'_\mu M^{\mu\nu} = q_\nu M^{\mu\nu} = 0$$

Hint: use the stored amplitude `Ampl(e,ep)` to define the contractions with `q` and `qp`:

```
Global AmplQ = Ampl(q,ep);
Global AmplQP = Ampl(e,qp);
```

Hint: to simplify the calculation, different substitutions have to be applied to the same quantities, such as `ps` and `pu`, depending on whether they enter `AmplQ` or `AmplQP`

Try to program it so that both `AmplQ` and `AmplQP` can be checked simultaneously!