SIX

Disclaimer: I do not understand all the mathematics here. Just the problem of the series expansions.

In this lecture we are going to study a complicated power series expansion with many variables and relations between them that cannot be applied immediately. This problem arises in the evaluation of the Euler characteristic of the automorphism group of the free group of rank n. The paper that set this off is "Moduli of graphs and automorphisms of free groups" by Marc Culler and Karen Vogtmann, https://link.springer.com/article/10.1007/BF01388734. This Euler characteristic is an interesting invariant (i.e. a number) of these groups and it is a tough problem to compute it. For instance, this is certified by the impressive paper of Shigeyuki Morita, Takuya Sakasai and Masaaki Suzuki with the simple abstract "We show that the integral Euler characteristic of the outer automorphism group of the free group of rank 11 is -1202." Archiv.1405.4063, Experimental Mathematics 24, (2015), 93-97. In this paper the authors made heavy use of a supercomputer to compute this value element by element. There were, if I understand it right, about 4.9 10^{12} elements to be calculated.

The next step in the evaluation of these Euler characteristics was made by Michael Borinsky in collaboration with Karen Vogtmann. That paper will come out soon. In this paper the authors find a way to compute this Euler characteristic without having to consider all individual cases separately. In the words of Michi (Michael Borinsky):
"The trick is to formulate this series of numbers (the Euler characteristic of $\mathrm{Out}(F_n)$) as a certain 0-dimensional quantum field theory with infinitely many interacting fields. The Lagrangian of this QFT is the one given in the initial stage of the program. The program does nothing but going from this 0-dim Lagrangian to the associated partition function by expanding the (0-dim) path integral (and there is the extra step of going to this twisted free energy in the end.)"

As is usual with mathematics papers it takes a while to write things down properly. But Michi (Michael Borinsky) had the main method already in 2020 and tried to obtain the series expansion in which the 10-th element (chi=10) would be this Euler characteristic of $\mathrm{Out}(F_{11})$. At the same time he wanted to learn how to do this in FORM and see whether it was possible to obtain more elements in the expansion. It would not be fair to show his program here, because he was still learning. I can give some timings though (timings on an apple powerbook with a M1max processor):

chi	CPU time
4	3.80 sec
5	31.97 sec
6	217.08 sec
7	1398.72 sec
8	8665.82 sec

As you can see, this simple approach could probably do the chi=10 case in about 90 hours on this laptop (which did not exist in 2020).

This is the point where I got involved. We will start with the program that does the expansions with the use of the techniques we have seen already in the second and third lectures. That program can indeed obtain a few extra terms. Then we will keep improving it and see how many more terms we can obtain in this expansion. This is what we did together in the year 2020. As it turns out, the resulting program will also be applicable for more mathematics problems. This will be pointed out in the paper by MB and KV.

The formula we have to work out is

$$\begin{aligned} L_1(n,m) &= \sum_{j=1}^{m} g^j z^j / j (c_j + c_j^2/2 - \theta(n-2j) c_{2j}/2 + G(j,n)) \\ G(j,n) &= (1+c_j)(\sum_{k=1}^{\lfloor n/j \rfloor} \mu(k)/k \sum_{i=1}^{\lfloor n/k \rfloor} (-1)^i c^i_{k\ j}/i) \end{aligned}$$

Here μ is the Möbius function and the c_i obey the conditions:

$$\begin{aligned} c_j &= l^j z^{\lfloor -j/2 \rfloor} \quad j \text{ even and } j \geq n/2+1 \\ c_j &= 0 \qquad\quad j \text{ odd and } j \geq n/2+1 \\ c_j &\to l^j c_j \qquad \text{otherwise} \end{aligned}$$

We don't really need to keep the g. It is has 'weight' -2. Hence

$$g = 1/l^2$$

Next we see this as an expansion in l and have to work out the exponent of L_1. After this odd powers of l can

be eliminated and even powers set to one. At this point we have a formula in the c_i and z. Now is the moment we can use the relations for the c_i:
The even c_i get shifted: $c_{2i} \to c_{2i} + 1/z^i$. Once this has been done, odd powers of the c_i are eliminated. Finally we can substitute:

$$c_i^n = \frac{n!}{2^{n/2}(n/2)!}(\frac{i}{z})^{n/2}$$

The resulting formula is an expansion in negative powers of z of which we now need the logarithm after which only one step remains. The series we have is $\sum_{m=0}^{n} a_m z^{-m}$ and out of it we construct the new series

$$C = \sum_{j=1}^{n} \sum_{k=1}^{\lfloor n/j \rfloor} \frac{\mu(k)}{k} 1/z^{j\ k} a_j$$

The expression C is what we are after and its n-th coefficient is the integer Euler characteristic of the automorphism group of the free group F_{n+1}. This means that the 10-th coefficient should be -1202 if the Japanese paper and the MV/KV method are in agreement. Our task here is to see how far we can take this series.

First we have to make sure we can handle the Möbius function. It is defined by

- $\mu(n) = +1$ if n is a square-free positive integer with an even number of prime factors.
- $\mu(n) = -1$ if n is a square-free positive integer with an odd number of prime factors.
- $\mu(n) = 0$ if n has a squared prime factor.

In 2020 FORM did not have a Möbius function yet. Hence we had to evaluate the values we needed in the FORM program. For this we can use the formulas:

$$\begin{aligned} \mu(1) &= 1 \\ \sum_{k=1}^{n} \lfloor n/k \rfloor \mu(k) &= 1 \end{aligned}$$

```
#define chi "10"
#define n "{`chi'*6}"
CFunction mu;
Symbol j,x;
Local M = sum_(j,1,`n',mu(j)*x^j);
Bracket x;
.sort
#do j = 1,`n'
  id mu(`j') = 1-sum_(j,1,`j'-1,integer_(`j'/j)*M[x^j]);
  Bracket x;
  .sort
#enddo
```

```
Bracket x;
.sort
Hide M;
.sort
```

If we print the output of this program for chi=10 we obtain:

```
M =
   x - x^2 - x^3 - x^5 + x^6 - x^7 + x^10 - x^11 - x^13 + x^14 + x^15 -
   x^17 - x^19 + x^21 + x^22 - x^23 + x^26 - x^29 - x^30 - x^31 + x^33 +
   x^34 + x^35 - x^37 + x^38 + x^39 - x^41 - x^42 - x^43 + x^46 - x^47 +
   x^51 - x^53 + x^55 + x^57 + x^58 - x^59;
```

Because we bracket in x, the value of $\mu(n)$ is given by `M[x^n]`. This method is rather easy to program in FORM, but it is inherently quadratic. Hence in version 4.3 of FORM the original algorithm is used because internally FORM has a sufficiently large list of prime numbers. Its main limitation is however that it cannot handle arguments that are greater than 2^{62}, but that is anyway much bigger than the above algorithm would be able to handle realistically.

The first program follows more or less the techniques we have seen before. We slowly expand step by step and eliminate whatever can be eliminated as soon as possible. Let us have a look at the first program. It begins with

```
#-
#: WorkSpace 1G
* Calculates the integral Euler Characteristic of Out(F_n) where n = chi + 1
#define chi "10"
#define m "{`chi'*2}"
#define n "{`chi'*6}"
CFunction mu,F;
Symbol i,j,n,y,x,z(-`chi':`m'),w,g,l({-2*`m'}:`n');
* Moebius function:
* Def:
* mu(1) = 1
* \sum_{d divides n} \mu(d) = 0 if n > 1
*
* Code uses the identity:
* \sum_{k=1}^n \floor{n/k} \mu(k) = 1
Local M = sum_(j,1,`n',mu(j)*x^j);
#do j = 1,`n'
    id mu(`j') = 1-sum_(j,1,`j'-1,integer_(`j'/j)*M[x^j]);
```

```
    Bracket x;
    .sort
#enddo
Bracket x;
.sort
Hide M;
.sort
```

We start with defining chi as a preprocessor variable and in addition some auxiliary variables m and n. The symbols will be needed later on and we put some power restrictions that allow quick elimination of irrelevant terms at the later stages of the program. Next comes the Möbius function which is stored in the hide system, allowing us to access the coefficients of the powers of x.

Next we define our formula, together with some obvious simplifications that reduce the number of terms a bit:

```
Symbol x1,...,x‘n’;
Symbol c1,...,c‘n’;
Local L1 =
  #do j = 1,‘m’
    + g^‘j’ * z^‘j’ / ‘j’ * (
       c‘j’ + c‘j’^2/2
       #if {2*‘j’} <= ‘n’
```

```
          - c{2*‘j’} / 2
        #endif
       - (1+c‘j’) * (
         #do k = ‘j’,‘n’,‘j’
           + M[x^{‘k’/‘j’}]/{‘k’/‘j’} *
           sum_(j,‘k’,‘n’,‘k’,(-1)^(j/‘k’+1) * c‘k’^(j/‘k’)/(j/‘k’))
         #enddo
       )
     )
  #enddo
;
* Still some corner cutting:
#do j = ‘n’,{‘n’/2+1},-2
    id c{‘j’} = l^‘j’ * z^{-‘j’/2};
#enddo
#do j = ‘n’-1,{‘n’/2+1},-2
    id c{‘j’} = 0;
#enddo
#do j = 1,‘n’
    id c‘j’ = l^‘j’ * c‘j’;
#enddo
```

```
* We don't really need to keep the g. It is has 'weight' -2:
id g = l^(-2);
B l;
.sort
Hide L1;
```

One can see the use of the Möbius function. We have used here a rather direct definition of the formula that we have to work with. But for chi=10 it is already getting sizable as we can see in the output:

```
Time =       0.01 sec    Generated terms =        994
              L1         Terms in output =        683
                         Bytes used      =      25644

  L1 =

     + l * ( 1/3*z*c3 + 1/2*z*c1*c2 + 1/6*z*c1^3 )
     + l^2 * (  - 1/4*z*c2^2 + 1/3*z*c1*c3 - 1/12*z*c1^4 + 1/6*z^2*c6 + 1/4*
       z^2*c2*c4 + 1/12*z^2*c2^3 )
     + l^3 * ( 1/5*z*c5 - 1/4*z*c1*c2^2 + 1/20*z*c1^5 + 1/9*z^3*c9 + 1/6*z^3
       *c3*c6 + 1/18*z^3*c3^3 )
           .
           .
           .
     + l^58 * (  - 1/60*z*c30^2 + 1/60*z*c15^4 + 1/60*z*c10^6 + 1/60*z*c6^10
```

```
- 1/60*z*c5^12 - 1/60*z*c3^20 - 1/60*z*c2^30 - 1/3540*z*c1^60 );
```

We see the highest c to be c30 and the highest power of z is 20.

The next step is to take the exponent of this expression. We can do this with a slow expansion as we saw before, making one substitution at a time after which we sort to reduce the number of terms.

We start with determining which elements of L1 are needed, with the sidecondition that the sum of the numbers of the elements c_i must be even. This is done in this step

```
Local L = sum_(j,0,‘n’,x^j/fac_(j));
#do i = ‘n’,1,-1
  id,once,x^‘i’*l^n? = x^‘i’/x*l^n*sum_(j,1,‘m’-n,F(j)*l^j);
.sort:three-‘i’;
#enddo
id  l^n?odd_ = 0;
id  l = 1;
.sort:three;
```

In here $F(n)$ represents the contents of the l^n bracket in the expression L_1. The output is:

```
Time =       0.45 sec    Generated terms =       1538
               L         Terms in output =       1538
                   three Bytes used      =      48392

  L =

     + 1
```

```
+ 1/2*F(1)^2
+ 1/24*F(1)^4
+ 1/720*F(1)^6
+ 1/40320*F(1)^8
+ 1/3628800*F(1)^10
+ 1/479001600*F(1)^12
+ 1/87178291200*F(1)^14
+ 1/20922789888000*F(1)^16
+ 1/6402373705728000*F(1)^18
+ 1/2432902008176640000*F(1)^20
+ 1/6402373705728000*F(1)^18*F(2)
+ 1/355687428096000*F(1)^17*F(3)
      .
      .
      .
+ 1/2*F(10)^2
+ F(12)
+ F(14)
+ F(16)
+ F(18)
+ F(20)
```

```
    ;
```

This step takes already a little bit of time, but that is nothing compared to if we would have substituted the `F(n)` immediately. We make this substitution very carefully in the next step:

```
*
*   This is an expensive step. Specially if we do it this way:
*   id  F(n?) = L1[l^n];
*   The problem is that there are products of brackets and potentially
*   even high powers.
*
#do ii = 1,1
    #$success = 0;
    id,once,ifnomatch->nomatch,F(x?) = L1[l^x];
    if ( count(F,1) ) $success = 1;
    label nomatch;
    .sort
    #if ( `$success' == 1 )
        #redefine ii "0"
    #endif
#enddo
.sort:four;
```

The trick with the `#do ii = 1,1` together with the `#redefine ii "0"` creates effectively a repeat loop with a `.sort` inside. It is of course rather dirty to redefine the loop parameter, but it is the easiest way, considering that we cannot sort inside a repeat loop. This way we can take one occurrence of F at a time and sort, until there are no more occurrences of F. Even so, this is costly:

```
Time =       9.74 sec    Generated terms =    1157892
               L         Terms in output =    1157892
                    four Bytes used      =   62070372

  L =

     + 1
     + 1/60*z^-10
     + 1/54*z^-9
     + 1/48*z^-8
     - 1/216*z^-8*c2^2
     + 1/162*z^-8*c1*c3
     - 1/648*z^-8*c1^4
     + 1/42*z^-7
     - 1/2592*z^-7*c6
     - 19/7776*z^-7*c3^2
     + 1/216*z^-7*c2*c4
     - 1/192*z^-7*c2^2
```

```
 + 13/2592*z^-7*c2^3
 + 1/240*z^-7*c1*c5
 + 1/144*z^-7*c1*c3
 + 1/324*z^-7*c1*c2*c3
 + 1/432*z^-7*c1^2*c2^2
 + 1/972*z^-7*c1^3*c3
       .
       .
       .
 + 1/27866776976946070750966579200 0*z^20*c1^54*c2^3
 + 1/39013487767724499051353210880 00*z^20*c1^55*c2*c3
 + 1/52017983690299332068470947840 00*z^20*c1^56*c2^2
 + 1/22237688027602964459271330201 6000*z^20*c1^57*c3
 + 1/14825125351735309639514220134 4000*z^20*c1^58*c2
 + 1/88950752110411857837085320806 40000*z^20*c1^60
;
```

Also the number of terms is growing rapidly.

At this point we can start simplifying things with the c's. This could not be done earlier, because there are conditions that hold for the full product of those c's in each term.

*

```
*   In the next recursion we use the function num, because (n-1) at ground
*   level gives two terms, and if we go m times through the recursion we
*   will get 2^m terms. This way we keep it one term.
*   One can also work with factorials etc, but that does not gain much more.
*   This is simpler.
*
CF num;
#do j = `n',1,-1
* This step was also very expensive.
* It is faster if one cleans up right afterwards.
    #if {`j'%2} == 0
        id c`j' = x`j' + z^{-`j'/2};
    #else
        id c`j' = x`j';
    #endif
    id  x`j'^n?odd_ = 0;
    repeat;
        id x`j'^n?pos_ = num(n-1)*x`j'^n/x`j'^2/z^`j'*`j';
        id  num(x?) = x;
    endrepeat;
    .sort:five-`j';
```

```
#enddo
Bracket z;
Print +f;
.sort:six;
```

The trick with the function num makes that each time the argument of num is reduced to a single term and only after that we remove the num function. But now, thanks to all power restrictions we obtain just a power series in negative powers of z.

```
L =
    + z^-10 * (  - 1299 )
    + z^-9 * (  - 96 )
    + z^-8 * ( 13 )
    + z^-7 * ( 23 )
    + z^-6 * ( 16 )
    + z^-5 * ( 10 )
    + z^-4 * ( 6 )
    + z^-3 * ( 4 )
    + z^-2 * ( 2 )
    + z^-1 * ( 1 )
    + 1;
```

And now the final steps are rather trivial and give us the answer. First we take the logarithm and then we apply

the final formula with again the Möbius function:

```
L Log = sum_(j,1,‘chi’, (-1)**(j+1) * y^j/j );
#do j = 1,‘chi’
  id once y = L-1;
  .sort
#enddo
B z;
.sort
Hide Log;
L C = sum_(j,1,‘chi’, sum_(l,1,integer_(‘chi’/j),z^(-l*j) * M[x^l]/l *
                Log[z^(-j)]));
id  z^n? = z^-n; * looks nicer
Print +f +s;
.end
```

with the answer

```
  C =
     + z
     + z^2
     + 2*z^3
     + z^4
```

```
     + 2*z^5
     + z^6
     + z^7
     - 21*z^8
     - 124*z^9
     - 1202*z^10
    ;
  27.13 sec out of 27.13 sec
```

and indeed the 10-th term is -1202. This time after 27 sec on a (very good) laptop.

Does this end the project?

Of course not! This is an encouraging way to get started.

First we want to see how far we can take this program. This means that if we raise the value of chi we are in virgin territory, obtaining the Euler characteristic of much bigger groups.

chi	CPU time
10	27.13 sec
11	79.50 sec
12	206.31 sec
13	839.12 sec
14	5505.08 sec
15	13044.85 sec
16	29297.34 sec

with the answer

```
C =
     + z
     + z^2
     + 2*z^3
     + z^4
     + 2*z^5
     + z^6
     + z^7
     - 21*z^8
     - 124*z^9
     - 1202*z^10
```

```
  - 10738*z^11
  - 112901*z^12
  - 1271148*z^13
  - 15668391*z^14
  - 208214777*z^15
  - 2975254451*z^16
    ;
```

Until now we have not done anything special.(Of course MB and KV did!)

Now we make the observation that we have no noncommuting variables and hence we can split the exponents[1]:

$$e^{a+b} \;=\; e^{a}\; e^{b}$$

to create separate exponent calculations. This keeps the numbers of terms down, provided we split the expression in a proper way. We will create n expressions out of our complete expression with the condition that expression L_i will have only terms containing at least one c_i and no c_j with $j > i$. This is done easily in FORM:

```
*
* Now we have to set up a loop in which we work our way down from the
* high c‘i’ to the c1.
* This should be done slowly to avoid building up to many terms.
* Each time we try to eliminate the respective c‘i’ as quickly as possible.
*
On Statistics;
L L0 = LL;
.sort
#do i = ‘n’,1,-1
L L‘i’ = L0;
if ( expression(L‘i’) );
  if ( count(c‘i’,1) == 0 ) Discard;
else;
```

[1] Noncommuting variables and the Campbell Baker Hausdorf formula are treated in a separate chapter that has not been included in the course this year.

```
    id  c‘i’ = 0;
  endif;
  B+    l;
  .sort:L‘i’;
  Hide L‘i’;
  #enddo
  Hide L0;
```

At this point we can start constructing the various exponents. We start at the highest c_i and work down to c_1 and each time we have multiplied in one exponent we can use the relations for this highest c_i to simplify it. The result is that the expressions do not become extremely lengthy.

```
Hide L0;
L   L = LE;
*
*   Now the real stuff starts
*
#do i = ‘n’,1,-1
#if ( termsin(L‘i’) > 0 )
  #$pass = 0;
  #do ii = 1,1
    #$success = 0;
```

```
  #$pass = $pass+1;
  id,once,ifmatch->success,F(1)^3 =
              L‘i’[l]^3+3*L‘i’[l]^2*FF(1)+3*L‘i’[l]*FF(1)^2+FF(1)^3;
  id,once,ifmatch->success,F(1)^2 = L‘i’[l]^2+2*L‘i’[l]*FF(1)+FF(1)^2;
  id,once,ifmatch->success,F(x?)^2 = L‘i’[l^x]^2+2*L‘i’[l^x]*FF(x)+FF(x)^2;
  id,once,ifnomatch->nomatch,F(x?) = L‘i’[l^x]+FF(x);
  label success;
  if ( count(F,1) ) $success = 1;
  label nomatch;
  ModuleOption maximum,$success;
  .sort: pass‘i’-‘$pass’;
  #if ( ‘$success’ == 1 )
      #redefine ii "0"
  #endif
#enddo
Multiply replace_(FF,F);
#if ( {‘i’%2} != 0 )
  id c‘i’ = x‘i’;
#else
  id c‘i’ = x‘i’ + z^{-‘i’/2};
#endif
```

```
  id    x‘i’^n?odd_ = 0;
  id    x‘i’^n?pos_ = fac_(n)*invfac_(n/2)/2^(n/2)*‘i’^(n/2)/z^(‘i’*n/2);
  .sort:five-‘i’;
#endif
#enddo
*
*   Now there is still the L0.
*
id  F(n?) = L0[l^n];
Bracket z;
.sort:six;
```

Because the L0 still contains terms without c_i, we need to take it also into account. We also programmed some special powers that occur frequently to make use of binomial coefficients. The ModuleOption statement is ignored by regular (sequential) FORM. When you run TFORM there would be a problem with $variables when several threads might want to put a value for $success in the central administration. By telling TFORM that we are looking for a maximum TFORM can take this into account and will not upgrade the value if it is not bigger than the already stored value. The final part of the program is still the same as before. The timings become now:

chi	CPU time
10	4.57 sec
11	10.05 sec
12	21.30 sec
13	42.74 sec
14	85.82 sec
15	166.70 sec
16	312.62 sec
20	3363.77 sec

This program is very much faster. For the case of chi=16 almost 100 times!

Can we improve this even more?

YES

In the previous program we substituted the function F using a small amount of sophistication in the form of

```
id,once,ifmatch->success,F(1)^3 = L‘i’[l]^3+3*L‘i’[l]^2*FF(1)
                                  +3*L‘i’[l]*FF(1)^2+FF(1)^3;
id,once,ifmatch->success,F(1)^2 = L‘i’[l]^2+2*L‘i’[l]*FF(1)+FF(1)^2;
id,once,ifmatch->success,F(x?)^2 = L‘i’[l^x]^2+2*L‘i’[l^x]*FF(x)+FF(x)^2;
id,once,ifnomatch->nomatch,F(x?) = L‘i’[l^x]+FF(x);
```

but as we saw already in lesson 2, it might pay to start with the most complicated powers and give them a chance to generate lower powers that cancel other lower powers. For that we have to modify things a bit. The part in which we express the substition of a power series inside the expansion of the exponent becomes now

```
L   LE = sum_(j,0,‘n’,x^j*invfac_(j));
#do i = ‘n’,1,-1
  id,once,x^‘i’*l^n? = x^‘i’/x*l^n*sum_(j,1,‘m’-n,F(-1,j)*l^j);
  .sort:three-‘i’;
#enddo
On Statistics;
repeat id F(n1?,x?)*F(n2?,x?) = F(n1+n2,x);
id  l^n?odd_ = 0;
id  l = 1;
.sort:three;
```

At this point we get (for chi=10) an output like

```
Time =       0.53 sec    Generated terms =       1538
              LE         Terms in output =       1538
                         Bytes used      =      52980
   LE =
        + 1
        + 1/24329020081766400000*F(-20,1)
        + 1/6402373705728000*F(-18,1)
        + 1/6402373705728000*F(-18,1)*F(-1,2)
        + 1/355687428096000*F(-17,1)*F(-1,3)
        + 1/20922789888000*F(-16,1)
        + 1/41845579776000*F(-16,1)*F(-2,2)
        + 1/20922789888000*F(-16,1)*F(-1,2)
        + 1/20922789888000*F(-16,1)*F(-1,4)
        + 1/1307674368000*F(-15,1)*F(-1,2)*F(-1,3)
        + 1/1307674368000*F(-15,1)*F(-1,3)
                .
                .
        + F(-1,8)*F(-1,12)
        + F(-1,9)*F(-1,11)
        + F(-1,10)
        + F(-1,12)
```

```
 + F(-1,14)
 + F(-1,16)
 + F(-1,18)
 + F(-1,20)
;
```

in which the highest powers of an object come first. The inner loop where we substitute the F becomes now:

```
#do i = `n',1,-1
#if ( termsin(L`i') > 0 )
  #$pass = 0;
  #do ii = 1,1
    #$success = 0;
    #$pass = $pass+1;
    id,once,ifnomatch->nomatch,F(n?neg_,x?) = (L`i'[1^x]+FF(-1,x))^-n;
    repeat id FF(n1?,x?)*FF(n2?,x?) = FF(n1+n2,x);
    label success;
    if ( count(F,1) ) $success = 1;
    label nomatch;
    ModuleOption maximum,$success;
    .sort: pass`i'-`$pass';
    #if ( `$success' == 1 )
```

```
        #redefine ii "0"
      #endif
    #enddo
    Multiply replace_(FF,F);
    #if ( {‘i’%2} != 0 )
      id c‘i’ = x‘i’;
    #else
      id c‘i’ = x‘i’ + z^{-‘i’/2};
    #endif
    id  x‘i’^n?odd_ = 0;
    id  x‘i’^n?pos_ = fac_(n)*invfac_(n/2)/2^(n/2)*‘i’^(n/2)/z^(‘i’*n/2);
    .sort:five-‘i’;
  #endif
  #enddo
  *
  *   Now there is still the L0.
  *
  id  F(n?,x?) = L0[l^x]^-n;
```

We take the highest power of F in the highest ‘i’ once and leave the FF for lower values if ‘i’. We keep doing this until there are no more F’s. After that we rename FF into F and the outer loop will lower the value of ‘i’.

This approach changes the execution times into

chi	CPU time(old)	CPU time(new)
10	4.57 sec	4.38 sec
11	10.05 sec	9.54 sec
12	21.30 sec	19.32 sec
13	42.74 sec	38.04 sec
14	85.82 sec	75.52 sec
15	166.70 sec	142.43 sec
16	312.62 sec	261.86 sec
20	3363.77 sec	2489.58 sec

Although this improvement is not as spectacular as the previous one, it is significant.

In the next program we are going to improve the treatment of the Möbius function and the substitution of the c_i.

First we are going to put the Möbius function into a table. Table lookup is much faster than having to get parts of an expression. This is done with

```
Table mobius(1:‘n’);
Fill mobius(1) = 1;
#do j = 2,‘n’
 #$m = 1-sum_(j,1,‘j’-1,integer_(‘j’/j)*mobius(j));
 Fill mobius(‘j’) = ‘$m’;
#enddo
```

We make good use of the property that here the value of $m is evaluated inside the preprocessor. This way the table is filled with the numbers 0, 1, -1, and for each next element those simple values will be used. If we would have put

```
Table mobius(1:‘n’);
Fill mobius(1) = 1;
#do j = 2,‘n’
 Fill mobius(‘j’) = 1-sum_(j,1,‘j’-1,integer_(‘j’/j)*mobius(j));
#enddo
```

the table entries would be the RHS formulas that would have to be evaluated recursively whenever a table element is used and that would take an enormous amount of time.

The other improvement is to get a single formula for powers of the even c_i. Instead of

```
#if ( {‘i’%2} != 0 )
  id c‘i’ = x‘i’;
#else
  id c‘i’ = x‘i’ + z^{-‘i’/2};
#endif
id  x‘i’^n?odd_ = 0;
id  x‘i’^n?pos_ = fac_(n)*invfac_(n/2)/2^(n/2)*‘i’^(n/2)/z^(‘i’*n/2);
.sort:five-‘i’;
```

we use

```
#if ( {‘i’%2} != 0 )
  id  c‘i’ = x‘i’;
  id  x‘i’^n?odd_ = 0;
  id  x‘i’^n?pos_ = fac_(n)*invfac_(n/2)/2^(n/2)*‘i’^(n/2)/z^(‘i’*n/2);
#else
  id  c‘i’^n? = fac_(n)*sum_(j,0,integer_(n/2),invfac_(n-2*j)*
                invfac_(j)/2^j*‘i’^j)
                /z^({‘i’/2}*n);
#endif
.sort:five-‘i’;
```

This may be a small change, but it does improve things again:

chi	CPU time(old)	CPU time(new)
10	4.38 sec	3.94 sec
11	9.54 sec	8.41 sec
12	19.32 sec	17.23 sec
13	38.04 sec	34.05 sec
14	75.52 sec	67.17 sec
15	142.43 sec	127.69 sec
16	261.86 sec	235.93 sec
20	2489.58 sec	2217.31 sec

An improvement of more than 10%.

For the next improvement we first put the parts in which we substitute the i-th formula in the exponent and then eliminate the corresponding c_i into a procedure. This makes things easier to oversee. Next we put the formulas for the c_i inside a special function num. The advantage of this is that this is just a polynomial in z and its evaluation does not involve all the other parts of the terms. In addition we declare num to be a zero-dimensional table with a single argument. The fill statement then makes that num(x) is automatically replaced by x after the argument has been evaluated.

Hence:

```
Table num(x?);
Fill num = x;
```

and

```
#procedure reduction(i,par)
#if ( termsin(L‘i’) > 0 )
  Multiply replace_(FF,F);
  #$pass = 0;
  #do ii = 1,1
    #$success = 0;
    #$pass = $pass+1;
    #if ( ‘par’ == 0 )
      id,once,ifnomatch->nomatch,F(n?neg_,x?) = (L‘i’[1^x]+FF(-1,x))^-n;
```

```
    repeat id FF(n1?,x?)*FF(n2?,x?) = FF(n1+n2,x);
  #else
    id,F(n?neg_,x?) = (L`i'[1^x])^-n;
  #endif
  label success;
  if ( count(F,1) ) $success = 1;
  label nomatch;
  ModuleOption maximum,$success;
  .sort: pass`i'-`$pass';
  #if ( `$success' == 1 )
      #redefine ii "0"
  #endif
#enddo
#if ( `i' != 0 )
B c`i';
.sort:bracket-c`i';
Keep Brackets;
#if ( {`i'%2} != 0 )
  id  c`i'^n?odd_ = 0;
  id  c`i'^n?pos_ = num(fac_(n)*invfac_(n/2)/2^(n/2)*
                            `i'^(n/2))/z^(`i'*n/2);
```

```
  #else
    id  c‘i’^n? = num(fac_(n)*sum_(j,0,integer_(n/2),invfac_(n-2*j)*
                  invfac_(j)/2^j*‘i’^j))
                  /z^({‘i’/2}*n);
  #endif
  #endif
  .sort:remove-c‘i’;
#endif
#endprocedure
```

How do we use this now? After generating the L_i we use the code:

```
On Statistics;
L L = LE;
#call reduction(0,0)
#do ij = ‘n’,2,-1
  #call reduction(‘ij’,0)
#enddo
#call reduction(1,1)
Print +f;
Bracket z;
.sort:six;
```

This time we start with the L_0 and then we work our way down from the high i to the low i. The case for c_1 is treated a little bit special as we can see in the procedure. Because we made this procedure, we can call it several times for different cases without making the program extremely lengthy.

chi	CPU time(old)	CPU time(new)
10	3.94 sec	2.64 sec
11	8.41 sec	5.65 sec
12	17.23 sec	11.69 sec
13	34.05 sec	23.46 sec
14	67.17 sec	47.08 sec
15	127.69 sec	90.65 sec
16	235.93 sec	173.93 sec
20	2217.31 sec	1849.84 sec

The trick with the num function has gained us a bit, although not an excessive amount.

We can gain still a little bit more by putting the L_i in a table. As mentioned before: table lookup is much faster than getting information from expressions. We do this with the FillExpression statement that we saw in the lecture on 10-dimensional gravity. In this case though the argument in L_i is a symbol and that causes the powers of the symbol to define the table elements, provided of course that the expressions are bracketted in terms of this symbol.

```
L L0 = LL;
.sort
#do i = `n'/2,1,-1
L L`i' = L0;
if ( expression(L`i') );
 if ( count(c`i',1) == 0 ) Discard;
else;
 id c`i' = 0;
endif;
B+ l;
Print +f +s;
.sort:L`i';
Table,zerofill,Ltab`i'(0:`n');
FillExpression Ltab`i' = L`i'(l);
Hide L`i';
```

```
#enddo
B l;
.sort
Table,zerofill,Ltab0(0:‘n’);
FillExpression Ltab0 = L0(l);
Hide L0;
.sort
```

This results in:

chi	CPU time(old)	CPU time(new)
10	2.64 sec	2.48 sec
11	5.65 sec	5.63 sec
12	11.69 sec	11.31 sec
13	23.46 sec	23.16 sec
14	47.08 sec	46.41 sec
15	90.65 sec	88.52 sec
16	173.93 sec	169.20 sec
20	1849.84 sec	1790.10 sec

As one can see, there is a slight improvement again.

A bigger improvement is obtained when we consider the PolyFun feature of FORM. In the chapter on integration by parts we saw already the polyratfun which serves as a coefficient for terms. It is a function with two arguments which are considered to be a numerator and a denominator of a rational polynomial. The simpler variety of this is the polyfun. It has only a single argument and that argument is considered the coefficient of the term. If we put all powers of z inside the polyfun Fz we will have considerably fewer terms to deal with. And FORM knows that

```
Fz(z)+2*Fz(z^2+z) -> Fz(3*z+2*z^2)
```

etc. At the moment we are ready for it we will return to writing out the terms. Hence we need to declare

```
CFunction Fz;
PolyFun Fz;
```

and when we do not need any longer it we put

```
PolyFun;
id  Fz(x?) = x;
```

Next we notice that for the higher values of i there are few occurrences of each c_i and few products. This means that we can gain some time by doing a few steps together while i is large. For this we create another procedure:

```
#procedure rangereduction(imin,imax)
*
*   Similar to the reduction procedure, but now it eliminates a range
```

```
*   of c‘imin’ to c‘imax’ parameters in one go.
*   This is useful in the beginning of the reductions when the overhead
*   of the sorting is enormous compared to the amount of action needed for
*   the elimination of a single variable.
*
  Multiply replace_(FF,F);
  #$pass = 0;
  #do ii = 1,1
    #$success = 0;
    #$pass = $pass+1;
    id,once,ifnomatch->nomatch,F(n?neg_,x?) =
                (<Ltab‘imin’(x)>+...+<Ltab‘imax’(x)>+FF(-1,x))^-n;
    repeat id FF(n1?,x?)*FF(n2?,x?) = FF(n1+n2,x);
    label success;
    if ( count(F,1) ) $success = 1;
    label nomatch;
    ModuleOption maximum,$success;
    .sort: pass‘imin’/‘imax’-‘$pass’;
    #if ( ‘$success’ == 1 )
        #redefine ii "0"
    #endif
```

```
  #enddo
   B    <c‘imin’>,...,<c‘imax’>;
   .sort:bracket-c‘imin’-c‘imax’;
   Keep Brackets;
   #do i = ‘imin’,‘imax’
   #if ( {‘i’%2} != 0 )
    id  c‘i’^n?odd_ = 0;
    id  c‘i’^n?pos_ =
        num(fac_(n)*invfac_(n/2)/2^(n/2)*‘i’^(n/2))*Fz(1/z^(‘i’*n/2));
   #else
    id  c‘i’^n? = num(fac_(n)*sum_(j,0,integer_(n/2),invfac_(n-2*j)*
                  invfac_(j)/2^j*‘i’^j))
                  *Fz(1/z^({‘i’/2}*n));
   #endif
   #enddo
  .sort:remove-c‘imin’-c‘imax’;
#endprocedure
```

This routine deals with a range from imin to imax (inclusive). Note also the use of Fz.

It is now important to figure out with what kind of ranges to call this procedure. After some experimentation we settle on:

```
L L = LE;
#call reduction(0,0)
#call rangereduction({‘n’/4+1},{‘n’/2})
#call rangereduction({‘n’/8+1},{‘n’/4})
#call rangereduction({‘n’/12+1},{‘n’/8})
#call rangereduction({‘n’/16+1},{‘n’/12})
#do ij = ‘n’/16,2,-1
  #call reduction({‘ij’},0)
#enddo
#call reduction(1,1)
.sort
PolyFun;
id    Fz(x?) = x;
```

The times become now:

chi	CPU time(old)	CPU time(new)
10	2.48 sec	1.58 sec
11	5.63 sec	3.35 sec
12	11.31 sec	6.66 sec
13	23.16 sec	13.17 sec
14	46.41 sec	26.31 sec
15	88.52 sec	49.69 sec
16	169.20 sec	89.02 sec
20	1790.10 sec	901.36 sec

This is a significant improvement! Most of it is due to the use of the PolyFun.

The next improvement is that we work with the square root of z, called zsq to avoid powers in which we have to calculate '$n'/2$ etc. In addition we notice we can scale the c_i with zsq^i and hence simplify the formulas later in the program.

An additional improvement in this step is that we count how many occurrences there are of the function F in each term and put the maximum for the expression in the variable \$Fcount. This way we can end the loops in the reduction and rangereduction procedures a bit more efficiently.

chi	CPU time(old)	CPU time(new)
10	1.58 sec	1.47 sec
11	3.35 sec	3.15 sec
12	6.66 sec	6.28 sec
13	13.17 sec	12.47 sec
14	26.31 sec	25.06 sec
15	49.69 sec	47.22 sec
16	89.02 sec	84.92 sec
20	901.36 sec	840.43 sec

Again a more than 6% reduction in CPU time.

Up to now we have seen steady improvements, some large, some a bit smaller. But clearly we can now calculate this Euler characteristic way beyond what is known. To improve the program further though, we have to come up with some new ideas. The major new idea is to tabulate the exponents of the L_i. And to facilitate the later multiplications we take the polyfun Fz away again.

```
*
*   Here we take the exponent of the L‘i’ (or Ltab‘i’).
*   It can become a bit slow, but not as bad as the next step.
*
On Statistics;
#do i = 0,‘n’/2
  L   G‘i’ = sum_(j,0,‘m’,x^j*invfac_(j));
  #do ii = ‘m’,1,-1
    id,once,x^‘ii’*l^n? = x^‘ii’/x*l^n*sum_(j,1,‘m’-n,Ltab‘i’(j)*l^j);
    .sort:exponent-‘i’-‘ii’;
  #enddo
  id  l^n? = 1;
  B+  zsq;
  .sort
  Table,zerofill,Gtab‘i’(0:{2*‘chi’});
  FillExpression Gtab‘i’ = G‘i’(zsq);
```

```
  Hide G‘i’;
  .sort
#enddo
```

After this the most expensive step becomes the multiplications of the exponents. But here we can take the powers of zsq into account:

```
L   L = G0;
#do i = ‘n’/2,1,-1
  id  zsq^n? = zsq^n*sum_(j,0,2*‘chi’-n,zsq^j*Gtab‘i’(j));
  B   c‘i’;
  .sort
```

and the loop terminates in the familiar way with

```
*
*   The bracket in c‘i’ and the keep brackets make that we have
*   to evaluate each c‘i’^n only once. Hence this step is rather fast.
*
  Keep Brackets;
  #if ( {‘i’%2} != 0 )
    id  c‘i’^n?odd_ = 0;
    id  c‘i’^n?pos_ =
```

```
        num(fac_(n)*invfac_(n/2)/2^(n/2)*‘i’^(n/2));
  #else
    id  c‘i’^n? = num(sum_(j,0,integer_(n/2),fac_(n)*invfac_(n-2*j)*
                  invfac_(j)*{‘i’/2}^j));
  #endif
  B   z;
  .sort:remove-c‘i’;
#enddo
id  zsq^2 = z;
Print +f;
Bracket z;
.sort:z-expansion;
```

The final steps are the same as before and the improvement is spectacular:

chi	CPU time(old)	CPU time(new)
10	1.47 sec	0.20 sec
11	3.15 sec	0.30 sec
12	6.28 sec	0.45 sec
13	12.47 sec	0.67 sec
14	25.06 sec	0.99 sec
15	47.22 sec	1.43 sec
16	84.92 sec	2.04 sec
20	840.43 sec	7.36 sec

To have to look up the Möbius function either as a bracket in an expression or to have to construct a table for it, is not very efficient. But at the moment we started this project FORM did not have an internal Möbius function. At this point it becomes time to do something about it, because such functions can be applied in many problems. In addition it is a type of function that can be inserted easily in the FORM source code because it has an integer number for its output and it can be evaluated for positive integers. The algorithm makes use of the fact that FORM also keeps a list of prime numbers up to the highest prime number that is less than 2^{31}. This suggests a different algorithm than we used in the FORM code, because the algorithm we used externally is quadratic in nature, while the density of prime numbers is $1/\ln(n)$. Hence an algorithm that is a bit like the sieve of Eratosthenes goes more like $n\ln(n)$ when we build up the table. This gives C code that looks roughly like:

```
if ( AR.moebiustable[nn] != UNDEFINED )
    return((WORD)AR.moebiustable[nn]);
n = nn;
mu = 1;
if ( n == 1 ) goto putvalue;
for ( i = 0; i < AR.numinprimelist; i++ ) {
    x = AR.PrimeList[i];
    if ( n % x == 0 ) {
        n /= x;
```

```
            if ( n % x == 0 ) { mu = 0; goto putvalue; }
            if ( AR.moebiustable[n] != UNDEFINED ) {
                mu = -AR.moebiustable[n]; goto putvalue;
            }
            mu = -mu;
            if ( n == 1 ) goto putvalue;
        }
        if ( n < x*x ) break;
    }
    mu = -mu;
putvalue:
    AR.moebiustable[nn] = mu;
    return((WORD)mu);
```

In the actual code the prime number 2 is taken as a special case.

Remark: There will be a workshop in April 2023 on how to make improvements to the FORM source code, including functions like the Möbius function and much more.

End of intermezzo.

The final improvement involves as a starter the use of the moebius_ function to make the program a bit shorter. The major improvement though is that this time we tabulate the powers of the c_i that are needed. This means of course that we have to determine for each c_i its maximum power. For now we leave the tables of the G_i out. This gives us:

```
On Statistics;
#do i = 0,`N'/2
L   G`i' = sum_(j,0,{`M'/`$minl`i''},x^j*invfac_(j));
#do ii = `M'/`$minl`i'',1,-1
    id,once,x^`ii'*l^n? = x^`ii'/x*l^n*
        sum_(j,1,{`M'-`$minl`i''*(`ii'-1)}-n,Ltab`i'(j)*l^j);
.sort:G`i'-`ii';
#enddo
id  l^n? = 1;
B+  zsq;
.sort
Hide G`i';
.sort
#enddo
*
*   Here we set up tables for the powers of the c`i'.
```

```
*
#do i = ‘N’/2,1,-1
Table,zerofill,Ctab‘i’(1:{‘N’/‘i’});
#do in = 1,‘N’/‘i’
  #if ( {‘i’%2} != 0 )
    #if ( {‘in’%2} != 0 )
      #$t = 0;
    #else
      #$t = fac_(‘in’)*invfac_(‘in’/2)/2^(‘in’/2)*‘i’^(‘in’/2);
    #endif
  #else
    #$t = sum_(j,0,integer_(‘in’/2),
            fac_(‘in’)*invfac_(‘in’-2*j)*invfac_(j)*{‘i’/2}^j);
  #endif
  Fill Ctab‘i’(‘in’) = ‘$t’;
#enddo
#enddo
```

This is then used in such a way that whether we obtain the G_i from a file or from a table makes not much difference.

```
*
```

```
* The idea here is to multiply in such a way that we can
* substitute the powers of c‘i’ immediately. To avoid too much
* work we bracket in c‘i’ and zsq.
* Yet the number of terms generated is enormous, specially compared
* to the number of terms remaining after sorting.
* The worst step is for c6.
*
L L = G0;
#do i = ‘N’/2,1,-1
B zsq,c‘i’;
.sort:remove-c{‘i’+1};
Keep Brackets;
id zsq^n? = zsq^n*sum_(j,0,2*‘chi’-n,zsq^j*G‘i’[zsq^j]);
id c‘i’^n?pos_ = Ctab‘i’(n);
#enddo
id zsq^2 = z;
Print +f;
Bracket z;
.sort:z-expansion;
```

This improvement is even better than the previous tabulation:

chi	CPU time(old)	CPU time(new)
10	0.20 sec	0.09 sec
11	0.30 sec	0.14 sec
12	0.45 sec	0.19 sec
13	0.67 sec	0.28 sec
14	0.99 sec	0.41 sec
15	1.43 sec	0.58 sec
16	2.04 sec	0.83 sec
20	7.36 sec	2.90 sec
25		11.34 sec
30		38.48 sec
35		121.91 sec
40		355.87 sec
45		967.64 sec
50		2486.18 sec

This method gives us the best results we obtained in this project. And in addition, the program can be used for more problems of this nature. In fact the paper of MB and KV has not come out yet, because they found another problem to solve this way and want to include that in their paper as well. It seems to be a matter of what Lagrangian you start with.

Note that the improvement from the first program to the last program for the case chi=16 is 29297.34 sec $\rightarrow$ 0.83 sec.

Just out of curiosity we may try to put the G_i in tables again. On the M1max it gives a slight improvement, but on the Lenovo computer on which we run TFORM in some cases with a very high value of chi the results are actually a little bit worse. Hence we will run without the G_i tables.

At a given point one may run out of inspiration to find better algorithms or implementations. I am not claiming that the last program is perfect and if you manage to improve upon it, please let me know. But there are other ways to go beyond what we have at the moment. We can go to a bigger computer and let it run for a period that would not be practical on a laptop. But there is yet another way: TFORM.

Most modern computers have a multitude of cores. Can we use the combined power of these cores to speed up our programs? The answer to this question is yes, because the architecture of FORM is such that most programs do not even need modification and a few need minor modifications to run much faster on several cores than on a single core. How does this work?

As mentioned in the first lecture, in each module FORM reads an input expression and processes its terms one by one, after which the resulting terms are sorted and written to the output file. All FORM needs to do is to distribute those input terms over the various cores (threads) and when each thread has sorted its output, to merge these outputs into a complete output that goes to the output file. Whatever is in between can remain the same. That is of course the principle. In practise there are some things that the user should take into consideration. There is a bottleneck in this system: the master process has to read the input and divide the terms over the workers and has to do the very final stage of the merge to write the output. In addition the master process does the compilations. This means that at some stage only a single processor is working and we would like to minimise this period.

The first observation is that distributing the input term by term costs too much overhead in sending messages to the threads. Hence we work in buckets with typically some 500-1000 terms each. In addition, when the threads

are working, the master prepares already a new set of buckets, which means that when a thread signals that it has finished its bucket the master can give it immediately a pointer to a new full bucket. The bucket size can be set as one of the setup parameters (see the manual), but nearly always the default setting is close to optimal.

There are exceptions to the above bucket system: when there are brackets and a bracket index, the master just passes the location of a bracket to the workers and they have to read the terms themselves.

It can also happen that all terms that consume much execution time are sitting in the same bucket. In that case all workers would have to wait for the one with the bad bucket. To avoid this, when workers become idle, the master starts stealing terms back from that bad bucket and give them to idle workers.

The other bottleneck is in the sorting. Imagine that you use 32 workers. If all terms are different there will be 5 compare operations per term to merge the sorted results of each worker. If the master has to do that all in a single merge operation this will be much more work than when we design a tree of extra workers (called sortbots) each of which takes input from two workers and creates one merged output. Further in the tree there are sortbots taking input from two other sortbots. Etc. The final merge involves just two streams of sorted terms and hence the master only has to do one compare per term. Starting at 4 workers this makes execution faster at the cost of the use of more memory for intermediate cash buffers.

Taking all the above into account, we do our 'production runs' on a nice computer with very much memory, many cores and the possibility to use swap space, because usually big parts of the memory is left unused for large amounts of time and FORM always reads most of its memory sequentially. This is how nearly all big FORM calculations are done these days.

Let us see how that works out. We take the last version of our program and take it to a Lenovo computer that is 3 years old, has 64 cores, 1 Tbyte of memory and 22 Tbytes of SSD disks. We use actually only 32 cores, because we are not the only user. Also, beyond a certain, program dependent, point the efficiency of using more cores does not give corresponding improvements, because the overheads become a bigger factor. When you are running many different Feynman diagrams on such a computer it might be most efficient to run 4 diagrams simultaneously, each with 16 workers. We call this 4x16. Running 64x1 will give horrible traffic jams at the disk when inevitably many will be dealing with complicated graphs simultaneously, while 1x64 will cause potentially 64 workers all having to do disk-to-disk sorts at the same time, also causing traffic jams, although of a more gentile nature. For our problem we have only a single program. Hence there is no choice.

chi	CPU time	real time
40	2418.11 sec	125.04 sec
50	16452.56 sec	664.20 sec
60	89702.52 sec	3328.36 sec
70	454416.54 sec	15422.30 sec
80	1981770.27 sec	64599.45 sec
90	8682409.75 sec	278743.92 sec
100	34107536.40 sec	1088481.25 sec

It is actually a demonstration of how good the Apple M1max chip is when you compare these timings with TFORM with 8 workers on my laptop:

chi	CPU time	real time
50	3046.58 sec	421.09 sec
60	17624.40 sec	2402.48 sec
70	89071.49 sec	11497.79 sec

These shorter times are basically showing how much better the memory access times are on the M1max chip.

One may wonder whether it is possible to use a GPU. For instance the GPU part of the M1max can run up to 10.4 Tflops. Unfortunately FORM does not use floating point operations and in addition symbolic manipulation is very 'data moving intensive'. Hence we have never found a practical way to make use of the GPU, which is a great pity of course.

The run with chi=100 was a bit more than 12 days. We can say now:

We show that the integral Euler characteristic of the outer automorphism group of the free group of rank 101 is

- 517364218454502374839623546721783826121378541888886632118436091585133
  579165853212698923487537607633251476021285069776407303171308497331357
  469167366322874

The full output of the chi=100 program can be found in the website together with all the programs we ran. This looks like a good point to stop.