SIX

This project was originally a session of two hours, but here we have cut it in two sessions of one hour each. The second part is given as session eight.

In this session we will look at an example from 10 dimensional gravity. The problem was posed to me by Ivano Lodato and Nabamita Banerjee and the program was made in collaboration with them. For more information about the physics see:
"The fate of flat directions in higher derivative gravity" by Nabamita Banerjeea, Suvankar Dutta, Ivano Lodato. JHEP 1305 (2013) 027, e-Print: arXiv:1301.6773 [hep-th].

Gravity projects usually bring some problems with them:

- They work with upper and lower indexes.
- They need all their vector and tensor components to be written explicitly.
- They have objects with many indexes.

In all gravity problems we start with a metric tensor and as explained before, we have to specify all the tensor components. Because we will be working in 10 dimensions that means that we have to specify 100 components. Some of these components will be formulas with variables in them, and most of them will be zero. We will use a table to specify the metric tensor, and we will use a very special technique to fill the table. In principle we could fill this table with fill statements, but we will see many more tables in this project, some with thousands of non-zero elements and there we will not have much choice.

We start with a number of declarations and put them in a file declare.h so that we do not have to repeat them all the time. Every once in a while we will add some extra lines to that file. The last two lines of the file will always be

```
Format nospaces;
.global
```

As we have seen before, the .global means that all declarations will remain valid after a .store instruction.

We start the declare.h file with the declarations

```
*
*   File with declarations for 10-dimensional gravity in
*   session 5 of the FORM course.
*
Dimension 10;
Symbols d1,d2,L,mu,rho,scqmu;
Symbols sint1,cost1,sint2,cost2;
AutoDeclare index i;
AutoDeclare symbol x;
AutoDeclare CFunction t,w,T,five;
CTable,zerofill,G(1:10,1:10);
CTable,zerofill,GI(1:10,1:10);
*
Format nospaces;
.global
```

We see a few new types of declarations here. First there is the dimension statement. This sets the default dimension. The default dimension is the dimension that is assigned to indexes

when we do not specify any dimension during their declaration. Here we say that all indexes will be in 10 dimensions.

Next there are the autodeclare statements. They are comparable to the implicit statement in FORTRAN, but with more flexibility. In the C language there is no equivalent. If we specify

```
AutoDeclare index i;
```

all objects that FORM encounters which start with the character i and have not been declared previously, will be automatically declared to be an index. And of course they will have the default dimension, because we did not specify a dimension. One is not restricted to a single character here. One could for instance make another statement

```
AutoDeclare symbol ijk;
```

This gives no conflict. What will happen is that if an undeclared object has a name that starts with ijk, it will be declared a symbol, and if its name starts with an i but not with ijk, it will become an index. The more restrictive condition takes precedence. Hence in the third autodeclare statement we have the string 'five', and therefore if the name of a previously undeclared variable starts with five, it will be seen as a commuting function, but if a name starts with fi or just f and it does not start with five, it is just an undeclared variable.

We use the autodeclare usually when we want to use lots of variables of a given type, but do not know yet what their exact names will be, or how many there will be.

The next declarations are the table declarations. G will be the metric tensor with lower indexes and GI will be its inverse, or the tensor with upper indexes. After all

$$g^{\mu\nu} g_{\nu\rho} \;=\; \delta^{\mu}_{\rho}$$

with $\delta^{\mu}_{\rho} = 1$ when $\mu = \rho$ and zero otherwise.

The declarations of the tables contain the option zerofill. This means that all elements of the table that have not been defined at the moment the table is used, will be assumed to be zero. This way we only have to specify the non-zero elements.

We start with the program

```
#-
#include declare.h
*
Local Fg =
      +tg(1,1)*(-d2/L^2+mu/d2)
      +tg(2,2)*(L^2*d2*rho^2/d1)
      +tg(3,3)*(L^2)
      +tg(4,4)*(L^2*sint1^2)
      +tg(5,5)*(L^2*cost1^2)
      +tg(6,6)*(L^2*cost2^2*sint1^2)
      +tg(7,7)*(L^2*sint1^2*sint2^2)
      +tg(8,8)*(d2)
      +tg(9,9)*(d2)
      +tg(10,10)*(d2)
      +(tg(1,5)+tg(5,1))*(-L*scqmu*cost1^2/d2)
      +(tg(1,6)+tg(6,1))*(-L*scqmu*cost2^2*sint1^2/d2)
      +(tg(1,7)+tg(7,1))*(-L*scqmu*sint2^2*sint1^2/d2)
```

```
        ;
Print +f;
Bracket tg;
.end
```

We define here an expression and we have the (commuting) function tg with two arguments to mark which element of the table they will eventually become. Note that tg will be declared as a commuting function due to an autodeclare statement.

The #- instruction turns off the listing of the input. For long programs that is often a good idea. With #+ one can turn it on again. How do we make a table from this? This is done by replacing the last three lines of the previous program by

```
Bracket tg;
.sort
Fillexpression G = Fg(tg);
    .
    .
```

The dots mean that we will add more code below. The FillExpression statement defines the contents of the table G by the contents of the brackets of the expression Fg, provided Fg has been bracketted in tg and the arguments of tg indicate the table elements to be specified.

This is a quick way to fill a whole table from an expression. The opposite can be obtained with the table_ function (see manual).

So now we have the metric tensor in a table and we have to determine the inverse of this tensor cq. matrix. Before we do this we should remark that there are some relations between the parameters in the metric tensor. They are

```
d2 = q + rho^2
d1 = d2^3 - rho^2*L^2*mu
scqmu^2 = q*mu
```

There are various ways by which we can compute the inverse of a matrix. One is by defining a matrix with 100 entries a_{ij}, multiply our matrix by it, setting the result equal to the unit matrix and solve the 100 equations. Another is by calculating minors and the determinant. Let us make an inventory to see how complicated it is to calculate the determinant. We could try to do this by writing all 10! terms, but because most elements of the matrix are zero we will be generating mainly zeroes. You will see we can do a whole lot better by using this fact.

Consider the following general purpose procedure

```
#procedure determ(F,T,N)
*
*   Routine evaluates the determinant of the NxN matrix
*   in table T. The result will be in expression F.
*   The method used is: Define the 1x1 minors in the
*   last column Then make the 2x2 minors from the last
*   two columns. Etc.    The minors are indicated by
*   the indexes in the Levi-Civita tensor e_. Hence
*   the coefficient of e_(2,3,5) is the minor in the
*   last three columns made from the entries in the
*   rows 2, 3 and 5. In this method no minor has to be
*   evaluated twice and no unneeded information is kept.
*   The trick with the 'Keep Brackets' makes that
*   zeroes are detected as quickly as possible.
*
```

```
Local ‘F’ = <e_(1)*‘T’(1,1)>+...+<e_(‘N’)*‘T’(1,‘N’)>;
#do k = 1,{‘N’-1}
  id  e_(i1?,...,i‘k’?) =
  #do i = 1,‘N’
    +e_(‘i’,i1,...,i‘k’)*‘T’({‘k’+1},‘i’)
  #enddo
    ;
  Bracket e_;
  .sort: determ at step ’k’;
  Skip;
  NSkip ‘F’;
  Keep Brackets;
#enddo
id  e_(1,...,‘N’) = 1;
#endprocedure
```

Because the routine is properly commented it should not be too difficult to figure out how it works. It is amazing how efficient it is.

```
.sort
FillExpression G = Fg(tg);
Drop;
.sort
#call determ(Fdet,G,10)
.sort
id scqmu^2 = mu*q;
repeat id d2^3 = d1+mu*L^2*(d2-q);
AntiBracket sint1,cost1,sint2,cost2;
Print +f +s;
.end
```

We add the above lines to our program and run it, curious how complicated the running will be. The answer is rather amazing:

```
Time =       0.00 sec    Generated terms =         11
            Fdet         Terms in output =          5
                         Bytes used      =        348

   Fdet=
```

```
+d1^-1*d2^2*L^12*mu*rho^2*q*(
   -sint1^6*cost1^2*sint2^2*cost2^2
   +sint1^6*cost1^4*sint2^2*cost2^2
   +sint1^8*cost1^2*sint2^2*cost2^4
   +sint1^8*cost1^2*sint2^4*cost2^2
   )
+d2^2*L^10*rho^2*(
   +sint1^6*cost1^2*sint2^2*cost2^2
   );
```

The other statistics, printed during evaluation of the determinant never have more than 17 terms generated in one module. It is hardly any work! The answer is still not something that we like to have in a denominator though. Inspection shows however that we could manipulate the sin's and the cos's a bit. How can we do that? If we just say

```
id  sint2^2 = 1-cost2^2;
id  sint1^2 = 1-cost1^2;
```

the spectator powers will blow up. We will have to be more sophisticated. We can do this by some typical FORM algorithms. Replace the last three lines by the following code (we have

put lots of print statements and .sort's to show what is happening).

```
AntiBracket sint1,cost1,sint2,cost2;
.sort
CFunction acc;
Collect acc;
Print +f +s;
.sort

Fdet=
   +acc(-sint1^6*cost1^2*sint2^2*cost2^2+sint1^6*cost1^4*sint2^2
   *cost2^2+sint1^8*cost1^2*sint2^2*cost2^4+sint1^8*cost1^2*
   sint2^4*cost2^2)*d1^-1*d2^2*L^12*mu*rho^2*q
   +acc(sint1^6*cost1^2*sint2^2*cost2^2)*d2^2*L^10*rho^2
   ;
```

The collect statement writes the contents of the brackets as arguments of the indicated function. Hence we have now two terms left.

```
Factarg acc;
```

```
 Print +f +s;
 .sort

Fdet=
   +acc(sint1,sint1,sint1,sint1,sint1,sint1,cost1,cost1,sint2,
   sint2,cost2,cost2)*d2^2*L^10*rho^2
   +acc(sint1,sint1,sint1,sint1,sint1,sint1,cost1,cost1,sint2,
   sint2,cost2,cost2,-1+cost1^2+sint1^2*cost2^2+sint1^2*sint2^2)
   *d1^-1*d2^2*L^12*mu*rho^2*q
   ;
```

FactArg will factorize the arguments of acc and write each factor as an argument of the function. Hence the acc function has now lots of arguments.

```
 ChainOut acc;
 Print +f +s;
 .sort

Fdet=
   +acc(sint1)^6*acc(cost1)^2*acc(sint2)^2*acc(cost2)^2*d2^2*
```

```
   L^10*rho^2
   +acc(sint1)^6*acc(cost1)^2*acc(sint2)^2*acc(cost2)^2*acc(-1+
   cost1^2+sint1^2*cost2^2+sint1^2*sint2^2)*d1^-1*d2^2*L^12*mu*
   rho^2*q
   ;
```

Chainout will write each argument of acc as a separate occurrence of the function acc with that argument.

```
id  acc(x?symbol_) = x;
id  acc(x?number_) = x;
Argument acc;
    id  sint1^2 = 1-cost1^2;
    id  sint2^2 = 1-cost2^2;
EndArgument;
Print +f +s;
.sort

Fdet=
   +d2^2*L^10*rho^2*sint1^6*cost1^2*sint2^2*cost2^2
```

```
    +acc(0)*d1^-1*d2^2*L^12*mu*rho^2*q*sint1^6*cost1^2*sint2^2*
    cost2^2
    ;
  id  acc(x?number_) = x;
  Print +f +s;
  .end

Fdet=
    +d2^2*L^10*rho^2*sint1^6*cost1^2*sint2^2*cost2^2
    ;
```

Each occurrence of acc that has only a single object can be written out again, and in the remaining occurrence we can now use the relations for sines and cosines. Lo and behold, it gives an argument zero! Hence in the end we have only a single term. This is very nice, because it means that our inverse matrix has only simple denominators.

Next we have to calculate the 100 minors. What we need is that when we calculate the (i,j) minor, rather than just substituting `G(i,j)` we should add `tgi(i,j)` and insist at the end that `tgi` be present only once. Hence we can modify the determ procedure so that it calculates simultaneously the determinant and all minors. This is done with:

```
#procedure minors(F,T,minor,N)
*
Symbol xinv(:1);
Local ‘F’ = <e_(1)*(‘T’(1,1)+‘minor’(1,1)*xinv)>+...+
          <e_(‘N’)*(‘T’(1,‘N’)+‘minor’(1,‘N’)*xinv)>;
#do k = 1,{‘N’-1}
id  e_(i1?,...,i‘k’?) =
#do i = 1,‘N’
    +e_(‘i’,i1,...,i‘k’)*(‘T’({‘k’+1},‘i’)
                          +‘minor’({‘k’+1},‘i’)*xinv)
#enddo
    ;
Bracket e_;
.sort: minors at step ’k’;
Skip;
NSkip ‘F’;
Keep Brackets;
#enddo
```

```
id  e_(1,...,`N') = 1;
.sort: minors completion;
if ( count(xinv,1) == 0 ) Multiply `minor'(0,0);
id  xinv= 1;
#endprocedure
```

We have stripped the commentary here as it would be nearly identical to the commentary in determ. The symbol xinv will make sure that no term will have more than one minor indicator. In the end the term that has no minor indicator is the determinant. We mark that as minor(0,0).

Also this routine is very fast, but simplifying the output using the relations between the variables is a bit more work. Hence it is best to make a procedure that does this in as general a way as possible. We will need that procedure many times for this project.

There are two things we need to simplify:

1. The sin/cos systems.

2. The d1/d2/rho/q systems.

In addition we have to take into account that there may be negative powers of some objects.

The first thing to do is to simplify the sin/cos system. If this would be the only simplification needed we can make the following procedure:

```
#procedure simsincos
*
*   Procedure simplifies combinations of sin and cos.
*   First we try 'at ground level'
*
id  cost1^2 = 1-sint1^2;
id  cost2^2 = 1-sint2^2;
*
AntiBracket sint1,cost1,sint2,cost2;
.sort: simsincos-1;
*
*   Now we collect the powers in a function acc.
*
Collect acc;
FactArg,acc;
Chainout,acc;
id  acc(-1+sint1)*acc(1+sint1) = -cost1^2;
id  acc(-1+sint2)*acc(1+sint2) = -cost2^2;
```

```
*
*   This is all we can do here.
*
id  acc(x?) = x;
.sort: simsincos-2;
#endprocedure
```

What we do here is first write everything to a unique form. Then we collect the occurrences of the sin/cos variables into the function acc and we factorize the arguments. A number of symbols and coefficients will be overal factors, but what we are really after is how to apply the relation $\sin^2 + \cos^2 = 1$. This means that we want to rewrite a factor $1 - \sin^2$ but not a facter $\sin^2$. Of course $1 - \sin^2$ will be factorized further. The ChainOut statement makes that those factors will be separate occurrences of the function acc. Hence the way the id-statements are written. Note that in expressions like

```
2-sint1^2+sint1^2*sint2^2
```

such simplifications cannot take place. On the whole the routine does however a decent job.

When we have a term with

```
acc(sint1^-2+cost1^-2-sint1^-2*cost1^-2)
```

it is pulled over a common denominator in the factarg statement. Hence the procedure will find this to be zero.

The more difficult procedure is the one that has to deal with the other relations. Because that is a somewhat more complicated system, it is much harder to have a decent result. In addition we may have negative powers in a more complicated way.

The proper thing to do is to pull all candidate terms over a common denominator. Hence the start of the main simplification procedure is

```
#procedure simd1d2
id  scqmu^2  = q*mu;
id  scqmu^-2 = 1/q/mu;
id  d1 = d2^3-rho^2*L^2*mu;
id  d2 = q+rho^2;
.sort: simd1d2-1;
#call simsincos
AntiBracket d1,d2,L,mu,q,rho,sint1,sint2,cost1,cost2;
.sort: simd1d2-2;
Collect acc;
#do d = {d1,d2,rho,L,sint1,cost1,sint2,cost2}
  $‘d’ = 0;
#enddo
Argument acc;
  #do d = {d1,d2,rho,L,sint1,cost1,sint2,cost2}
    if ( count(‘d’,-1) > $‘d’ ) $‘d’ = count_(‘d’,-1);
  #enddo
EndArgument;
```

```
Multiply 1
  #do d = {d1,d2,rho,L,sint1,cost1,sint2,cost2}
    *‘d’^$‘d’/d‘d’^$‘d’
  #enddo
    ;
id  acc(x?) = x;
id  q = d2-rho^2;
id  d2^3 = d1+rho^2*L^2*mu;
.sort: simd1d2-3;
#call simsincos
```

We start with trying to minimize the number of terms by writing out some relations. Next we hunt for simple combinations of sin/cos. Then comes the collection of the denominators: we antibracket in all variables that are part of the rewriting and put those brackets inside the function acc. Then we look for the maximum powers of the potential denominators. This is done with a special do-loop construction. If we define the loop with

```
#do d = {d1,d2,rho,L,sint1,cost1,sint2,cost2}
  $‘d’ = 0;
#enddo
```

the loop variable will take successively the string values d1, d2 etc. If one of the $-variables, say $d1, ends up positive, we multiply by `d1^$d1/dd1^$d1` and at the end of the procedure we will replace dd1 by d1. Because we have gotten new occurrences of the denominators in the numerators we need to apply some identities again. Then we look again in the sin/cos system. The remaining part of the procedure is

```
AntiBracket d1,d2,L,mu,q,rho;
.sort: simd1d2-4;
Collect acc;
FactArg acc;
ChainOut acc;
id  acc(x?number_) = x;
id  acc(x?symbol_) = x;
Argument acc;
    id q = d2-rho^2;
    repeat id d2^3 = d1+rho^2*L^2*mu;
EndArgument;
FactArg acc;
ChainOut acc;
```

```
id  acc(x?symbol_) = x;
id  acc(x?number_) = x;
Argument acc;
    id  d2*L^2*mu = rho^2*L^2*mu+q*L^2*mu;
    id  rho^2*L^2*mu = d2^3 - d1;
    if ( count(L,1,mu,1) == 0 ) id rho^2 = d2-q;
EndArgument;
FactArg acc;
ChainOut acc;
id  acc(x?symbol_) = x;
id  acc(x?number_) = x;
#call simsincos
#do d = {d1,d2,rho,L,sint1,cost1,sint2,cost2}
    id d‘d’^x? = ‘d’^x;
#enddo
#endprocedure
```

Note that once an occurrence of acc has only a simple argument, we take it out so that remaining substitutions inside acc do not spoil it anymore. Then there is some moving

around by means of identities to see whether we hit on single terms (which are taken out with the factarg/chainout and id-statements). Finally we try the sin/cos system again and substitute the denominators back.

With these procedures we are now ready to continue our project. First we compute the GI tensor which is the inverse of the G tensor. Hence the part after the definition of the G table is

```
FillExpression G = Fg(tg);
Drop;
.sort
#call minors(Finv,G,tgi,10)
.sort
#call simd1d2
Bracket tgi;
.sort
```

Here we compute the minors. The result is of course a bit messier than just computing the determinant. We obtain:

```
Finv=
   +tgi(0,0)*(d2^2*L^10*rho^2*sint1^6*cost1^2*sint2^2*
      cost2^2)
   +tgi(1,1)*(-d1^-1*d2^4*L^12*rho^2*sint1^6*cost1^2*sint2^2
      *cost2^2)
   +tgi(1,5)*(-d1^-1*d2^3*L^11*rho^2*scqmu*sint1^6*cost1^2*
      sint2^2*cost2^2)
   +tgi(1,6)*(-d1^-1*d2^3*L^11*rho^2*scqmu*sint1^6*cost1^2*
      sint2^2*cost2^2)
   +tgi(1,7)*(-d1^-1*d2^3*L^11*rho^2*scqmu*sint1^6*cost1^2*
      sint2^2*cost2^2)
   +tgi(2,2)*(d1*d2*L^8*sint1^6*cost1^2*sint2^2*cost2^2)
   +tgi(3,3)*(d2^2*L^8*rho^2*sint1^6*cost1^2*sint2^2*cost2^2
      )
   +tgi(4,4)*(d2^2*L^8*rho^2*sint1^4*cost1^2*sint2^2*cost2^2
      )
   +tgi(5,1)*(-d1^-1*d2^3*L^11*rho^2*scqmu*sint1^6*cost1^2*
      sint2^2*cost2^2)
```

```
+tgi(5,5)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2+d2^2*L^8*rho^2*sint1^6*sint2^2*cost2^2
   )
+tgi(5,6)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2)
+tgi(5,7)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2)
+tgi(6,1)*(-d1^-1*d2^3*L^11*rho^2*scqmu*sint1^6*cost1^2*
   sint2^2*cost2^2)
+tgi(6,5)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2)
+tgi(6,6)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2+d2^2*L^8*rho^2*sint1^4*cost1^2*sint2^2
   )
+tgi(6,7)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2)
+tgi(7,1)*(-d1^-1*d2^3*L^11*rho^2*scqmu*sint1^6*cost1^2*
   sint2^2*cost2^2)
```

```
+tgi(7,5)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2)
+tgi(7,6)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2)
+tgi(7,7)*(-d1^-1*d2^2*L^10*mu*rho^2*q*sint1^6*cost1^2*
   sint2^2*cost2^2+d2^2*L^8*rho^2*sint1^4*cost1^2*cost2^2
   )
+tgi(8,8)*(d2*L^10*rho^2*sint1^6*cost1^2*sint2^2*cost2^2)
+tgi(9,9)*(d2*L^10*rho^2*sint1^6*cost1^2*sint2^2*cost2^2)
+tgi(10,10)*(d2*L^10*rho^2*sint1^6*cost1^2*sint2^2*
   cost2^2);
```

As you can see, this is not very informative for now. It becomes worse for the bigger tensors we will be treating. Hence we will skip most of those outputs.

The inverse is now trivial:

```
Local Finv = Finv/(Finv[tgi(0,0)])-tgi(0,0);
Bracket tgi;
.sort
FillExpression GI = Finv(tgi);
Drop;
.sort
Local Fone = tone(i1,i3)*G(i1,i2)*GI(i2,i3);
Sum i1,1,...,10;
Sum i2,1,...,10;
Sum i3,1,...,10;
#call simd1d2
Bracket tone;
Print +f;
```

The invariant is stored in the GI table. Of course the definition of this table and all the variables we use in the procedures we just defined have to be declared in the file declare.h.

To make sure that we have indeed the inverse and $G_{ij}GI^{jk} = \delta_i^k$ we verify this relation. The tensor 'tone' is the marker for the position in the tensor: `tone(i,k)` is seen as the element

δ_i^k. We will use this in all tensors that we are going to define. Then we have to sum over the indexes. The statement

```
Sum i,1,...,10;
```

is equivalent to $\Sigma_{i=1}^{10}$ over the current term. This means that our three-fold sum generates in principle 1000 terms, but as soon as the G and GI are substituted, their zeroes will make most of those vanish. After simplification the output is indeed:

```
Fone=
   +tone(1,1)*(1)
   +tone(2,2)*(1)
   +tone(3,3)*(1)
   +tone(4,4)*(1)
   +tone(5,5)*(1)
   +tone(6,6)*(1)
   +tone(7,7)*(1)
   +tone(8,8)*(1)
   +tone(9,9)*(1)
   +tone(10,10)*(1);
```

At the final stages of this project we will have to evaluate 12-fold sums. The important thing is to do that in such a way that we do not need to evaluate 10^{12} terms. In the above program, what happens is that after the first two sums the values of G can be evaluated. Because this is a table, this is done automatically. This gives already many zeroes. As a result the third sum has to be applied to far fewer than 100 terms. This is what we will have to pay attention to.