FIVE

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. Jan 2013. 36 pp. 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 this term be present 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
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;
```

We compute the minors, simplify them and construct the inverse from them. This 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 submitted. 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.

Next are the connections. They are defined by

```
Local FGamma = +1/2*tgamma(i1,i2,i3)*GI(i1,i4)*der(i2,G(i4,i3))
               +1/2*tgamma(i1,i2,i3)*GI(i1,i4)*der(i3,G(i2,i4))
               -1/2*tgamma(i1,i2,i3)*GI(i1,i4)*der(i4,G(i2,i3));
```

with 'der' the derivative function. This means that we have to be able to take derivatives. For this we have to add the CFunctions 'der' and 'rem' to declare.h and we define two sets 'params' and 'vars':

```
Set vars:rho,sint1,cost1,sint2,cost2;
Set params:q,L,mu,scqmu,cqmu,u3;
```

with parameters that do and do not depend on the variables w.r.t. which we take derivatives. This makes life easier.

At this points we need some information about which parameters depend on which coordinates and how. This will be part of the procedure we need to write. This derivatives procedure is given by:

```
#procedure derivative
*
*   Procedure takes the derivative for our metric.
*
id  der(x?,0) = 0;
id  der(x?,xx?params) = 0;
id  der(x?{1,5,6,7,8,9,10},?a) = 0;
.sort: derivative-1;
SplitArg,der;
repeat id der(x1?,x2?,x3?,?a) = der(x1,x2)+der(x1,x3,?a);
FactArg,der;
id  der(x?,?a,x1?number_,?b) = der(x,?a,?b)*x1;
repeat id der(x?,?a,xx?params,?b) = der(x,?a,?b)*xx;
repeat id der(x?,?a,1/xx?params,?b) = der(x,?a,?b)/xx;
id  der(x?) = 0;
*
*   chain rule:
*
```

```
repeat;
    id  der(x?,x1?,x2?,?a) = der(x,x1)*rem(x2,?a)+x1*der(x,x2,?a);
endrepeat;
repeat id rem(x?,?a) = x*rem(?a);
id  rem = 1;
id  der(x1?,i_) = 0;
id  der(x1?,1/x2?vars) = -der(x1,x2)/x2^2;
id  der(x1?,1/d1) = -der(x1,d1)/d1^2;
id  der(x1?,1/d2) = -der(x1,d2)/d2^2;
id  der(2,rho)  = 1;
id  der(2,d1) = 6*rho*(rho^2+q)^2-2*rho*mu*L^2;
id  der(2,d2) = 2*rho;
id  der(3,sint1)= cost1;
id  der(3,cost1)= -sint1;
id  der(3,d1)=0;
id  der(3,d2)=0;
id  der(4,sint2)= cost2;
id  der(4,cost2)= -sint2;
```

```
id  der(4,d1)=0;
id  der(4,d2)=0;
id  der(x1?{2,3,4},x2?vars) = 0;
#endprocedure
```

First we note that derivatives w.r.t. anything but the coordinates 2,3,4 are zero. Then we apply that the derivative of a sum is the sum of the derivatives. Using factarg we split the arguments into individual symbols. We can take out the symbols that do not depend of the coordinates for which we take the derivatives. The chain rule needs an extra function rem which contains the remaining symbols. Then we define the derivatives of negative powers and we take into account that d1 and d2 depend on ρ, q and/or μ. Finally we substitute the derivatives. Anything not specified is zero.

The computation of the connections is now

```
Local FGamma = +1/2*tgamma(i1,i2,i3)*GI(i1,i4)*der(i2,G(i4,i3))
               +1/2*tgamma(i1,i2,i3)*GI(i1,i4)*der(i3,G(i2,i4))
               -1/2*tgamma(i1,i2,i3)*GI(i1,i4)*der(i4,G(i2,i3));
Sum i1,1,...,10;
Sum i4,1,...,10;
Sum i2,1,...,10;
Sum i3,1,...,10;
#call derivative
#call simd1d2
Bracket tgamma;
.sort
Fillexpression Gamma = FGamma(tgamma);
Drop;
```

This runs rather smoothly and our program gives the statistics

```
Time =       0.24 sec    Generated terms =          93
          FGamma         Terms in output =          93
                         Bytes used      =        7960
```

The next step is the Riemann tensor:

```
*
*   Set up the Riemann tensor.
*   Note that the first index is upper, the other three lower.
*
Local FRiemann =
        +triemann(i1,i2,i3,i4)*der(i3,Gamma(i1,i2,i4))
        -triemann(i1,i2,i4,i3)*der(i3,Gamma(i1,i2,i4))
        +triemann(i1,i2,i4,i3)*Gamma(i1,i4,i5)*Gamma(i5,i3,i2)
        -triemann(i1,i2,i3,i4)*Gamma(i1,i4,i5)*Gamma(i5,i3,i2);
if ( count(der,1) );
    Sum i1,1,...,10;
    Sum i2,1,...,10;
    Sum i4,1,...,10;
    id  der(?a,0) = 0;
    Sum i3,1,...,10;
else;
    Sum i1,1,...,10;
```

```
        Sum i5,1,...,10;
        Sum i4,1,...,10;
        Sum i2,1,...,10;
        Sum i3,1,...,10;
    endif;
    #call derivative
    #call simd1d2
    Bracket triemann;
    .sort
    FillExpression Riemann = FRiemann(triemann);
    Drop;
```

This is slightly more complicated because some terms have a derivative and some do not. We cannot do the sum over `i5` over the terms that do not have this index. Hence we need the if construction.

The statistics at this point are

```
Time =       0.35 sec    Generated terms =         742
       FRiemann          Terms in output =         742
                         Bytes used      =       66944
```

From the Riemann tensor we can construct the Ricci tensor:

```
Local FRicci  = tricci(i1,i2)*Riemann(i3,i1,i3,i2);
Sum i1,1,...,10;
Sum i2,1,...,10;
Sum i3,1,...,10;
#call simd1d2

Bracket tricci;
Print +f;
.sort
FillExpression Ricci = FRicci(tricci);
Drop;
```

Note that this is a contraction between an upper and a lower index. The remaining tensor has two lower indexes. The tensor has 22 components with in total 41 terms.

And of course we can now compute the curvature:

```
Local FR = tr*GI(i1,i2)*Ricci(i1,i2);
Sum i1,1,...,10;
Sum i2,1,...,10;
#call simd1d2
Bracket tr;
Print +f +s;
.sort
Hide FR;
```

Because we will need the curvature later we do not drop it but we put it in the hide system. A proper contraction needs an upper and a lower index, and hence we need to multiply by GI. It turns out that for this metric the curvature is zero. We are still at 0.36 sec.

The next step is to compute the Weyl tensor(s). The definition can be found in Wikipedia which gives two definitions. We take the second which has only lower indexes.

```
Local Fweyl0000 =
              tweyl(i1,i2,i3,i4)*(G(i1,i5)*Riemann(i5,i2,i3,i4)
             -1/8*Conv(Ricci,G,i1,i2,i3,i4)
             +R/80*Conv(G,G,i1,i2,i3,i4)
             -R/180*Conv(G,G,i1,i2,i3,i4));
id  Conv(f1?,f2?,i1?,i2?,i3?,i4?) =
             +f1(i1,i3)*f2(i2,i4)+f1(i2,i4)*f2(i1,i3)
             -f1(i1,i4)*f2(i2,i3)-f1(i2,i3)*f2(i1,i4);
.sort
if ( count(Riemann,1) );
    Sum i5,1,...,10;
endif;
Sum i2,1,...,10;
Sum i3,1,...,10;
Sum i4,1,...,10;
Sum i1,1,...,10;
```

```
.sort:weyl-3;
id  R = FR[tr];
#call simd1d2
Bracket tweyl;
.sort
FillExpression Weyl0000 = Fweyl0000(tweyl);
Drop;
```

We need the extra CFunctions Conv, f1 and f2. We will need to make many contractions involving Weyl tensors. Hence we cannot use the version with only lower indexes all the time. For convenience we adopt a notation in which we attach to the name a string of zeroes and ones that indicate whether the indexes are lower(0) or upper(1). This means that we have to add quite a few tables to the file declare.h. The statistics are now

```
Time =       0.50 sec    Generated terms =         864
      Fweyl0000          Terms in output =         864
                         Bytes used      =       80592
```

and we produce the next tensor with

```
Local Fweyl0001 = tweyl(i1,i2,i3,i4)*GI(i4,i5)*Weyl0000(i1,i2,i3,i5);
Sum i1,1,...,10;
Sum i2,1,...,10;
Sum i3,1,...,10;
Sum i5,1,...,10;
Sum i4,1,...,10;
#call simd1d2
Bracket tweyl;
.sort
FillExpression Weyl0001 = Fweyl0001(tweyl);
Drop;
```

Note the order in which we take the sums. We want to get to the zeroes in the Weyl tensor first. By similar techniques we produce also

```
Weyl1000   Weyl1100   Weyl1010    Weyl1001    Weyl0101
Weyl1110   Weyl1011   Weyl0111    Weyl1111
```

With the 11 tensors we have constructed this way we have enough for our project. The final statistics are

```
Time =       1.67 sec    Generated terms =       1540
       Fweyl1111         Terms in output =       1540
                         Bytes used      =     122944
```

Next we have to introduce something called the five-form. The five-form is a totally anti-symmetric tensor with 5 indexes. For this model it is given up to antisymmetrization by

```
Local Ffive00000 =
      +tfive(1,2,8,9,10)*( 4*rho*(rho^2+q)/L)
      +tfive(3,5,8,9,10)*( 2*L^2*cost1*sint1*scqmu)
      +tfive(3,6,8,9,10)*(-2*L^2*cost1*sint1*cost2^2*scqmu)
      +tfive(3,7,8,9,10)*(-2*L^2*cost1*sint1*sint2^2*scqmu)
      +tfive(4,6,8,9,10)*( 2*L^2*sint1^2*sint2*cost2*scqmu)
      +tfive(4,7,8,9,10)*(-2*L^2*sint1^2*sint2*cost2*scqmu)
        ;
```

We can antisymmetrize by (this use of the Levi-Civita tensor is very handy)

```
id  tfive(i1?,...,i5?) = e_(i1,...,i5)*e_(i6,...,i10)*
                         tfive(i6,...,i10);
Contract;
Bracket tfive;
.sort
FillExpression Five00000 = Ffive00000(tfive);
Drop;
```

Also of this tensor we will need varieties with different combinations of upper and lower indexes. As before

```
Local Ffive00001 = tfive(i1,i2,i3,i4,i5)*
                   Five00000(i1,i2,i3,i4,i6)*GI(i5,i6);
Sum i1,1,...,10;
Sum i2,1,...,10;
Sum i3,1,...,10;
Sum i4,1,...,10;
Sum i6,1,...,10;
Sum i5,1,...,10;
#call simd1d2
Bracket tfive;
.sort
FillExpression Five00001=Ffive00001(tfive);
Drop;
```

and we calculate as well

```
Five00011   Five00111   Five01111   Five11111
```

and for fun we compute

```
Local F55 = Five00000(i1,i2,i3,i4,i5)*Five11111(i1,i2,i3,i4,i5);
Sum i2,1,...,10;
Sum i3,1,...,10;
Sum i4,1,...,10;
Sum i5,1,...,10;
Sum i1,1,...,10;
#call simd1d2
Print +f +s;
.sort
Drop;
```

which gives as output

```
Time =       2.27 sec    Generated terms =          2
             F55         Terms in output =          2
                         Bytes used      =         84

   F55=
```

```
+960*d2^-3*mu*q
-1920*L^-2
;
```

The next step involves a conjugate

```
Local FHDfive00000=u3*L^5*tfive(i1,i2,i3,i4,i5)*
           ((1/120)*rho*(q+rho^2)*cost1*sint1^3*sint2*cost2)
           *e_(i1,...,i10)*Five11111(i6,...,i10);
Sum i6,1,...,10;
Sum i7,1,...,10;
Sum i8,1,...,10;
Sum i9,1,...,10;
Sum i10,1,...,10;
.sort
Sum i1,1,...,10;
Sum i2,1,...,10;
Sum i3,1,...,10;
Sum i4,1,...,10;
Sum i5,1,...,10;
id  e_(1,2,3,4,5,6,7,8,9,10)=i_;
#call simd1d2
Bracket tfive;
```

```
.sort
FillExpression HDfive00000=FHDfive00000(tfive);
```

and the tensor we are really after is

```
Local FCfive00000=tfive(i1,i2,i3,i4,i5)*
       (Five00000(i1,...,i5)+HDfive00000(i1,...,i5));
Sum i1,1,...,10;
Sum i2,1,...,10;
Sum i3,1,...,10;
Sum i4,1,...,10;
Sum i5,1,...,10;
#call simd1d2
Bracket tfive;
.sort
FillExpression CFive00000=FCfive00000(tfive);
Drop;
```

This tensor has a total 1800 terms. We construct now also

```
FCfive00001  FCfive00011  FCfive00111  FCfive01111  FCfive11111
```

At this point we can start to construct the tensors we need for the invariants we want to compute. We call this tensor T, it has 6 indexes and we will need a number of varieties with upper and lower indexes. When it has only lower indexes it is symmetric in the first 3 indexes vs. the last 3 indexes, and each of these groups of three indexes is totally antisymmetric between the three indexes. The definition, up to the symmetries, is

```
Local FT000000 =
    +i_*der(i1,CFive00000(i2,i3,i4,i5,i6))*tt(i1,i2,i3,i4,i5,i6)
    -i_*Gamma(i7,i1,i2)*(
         CFive00000(i7,i3,i4,i5,i6)*tt(i1,i2,i3,i4,i5,i6)
         CFive00000(i3,i7,i4,i5,i6)*tt(i1,i3,i2,i4,i5,i6)
         CFive00000(i4,i3,i7,i5,i6)*tt(i1,i4,i3,i2,i5,i6)
         CFive00000(i5,i3,i4,i7,i6)*tt(i1,i5,i3,i4,i2,i6)
         CFive00000(i6,i3,i4,i5,i7)*tt(i1,i6,i3,i4,i5,i2) )
    +1/16*CFive00000(i1,i2,i3,i7,i8)*CFive00011(i4,i5,i6,i7,i8)*
              tt(i1,i2,i3,i4,i5,i6)
    -3/16*CFive00000(i1,i2,i3,i7,i8)*CFive00011(i4,i5,i6,i7,i8)*
              tt(i1,i2,i6,i4,i5,i3)
    ;
```

Note that we have derivatives again. We work this out with

```
.sort
Sum i4,1,...,10;
Sum i5,1,...,10;
Sum i6,1,...,10;
.sort
if ( count(der,1) );
    Sum i2,1,...,10;
    Sum i3,1,...,10;
    id  der(x?,0) = 0;
    id  der(i?,0) = 0;
elseif ( count(Gamma,1) );
    Sum i3,1,...,10;
    Sum i7,1,...,10;
else;
    Sum i7,1,...,10;
    Sum i8,1,...,10;
endif;
```

```
.sort
if ( count(Gamma,1) );
    Sum i1,1,...,10;
    Sum i2,1,...,10;
elseif ( count(der,1) );
    Sum i1,1,...,10;
else;
    Sum i1,1,...,10;
    Sum i2,1,...,10;
    Sum i3,1,...,10;
endif;
.sort
#call derivative
#call simd1d2
.sort
```

This gets the tensor, but it is not in its symmetric form. We take care of that with

```
*
*   Now we impose the symmetries.
*
id  tt(x1?,x2?,x3?,?a) = (tt(x1,x2,x3,?a)-tt(x2,x1,x3,?a)
                         +tt(x2,x3,x1,?a)-tt(x3,x2,x1,?a)
                         +tt(x3,x1,x2,?a)-tt(x1,x3,x2,?a))/6;
.sort
id  tt(?a,x1?,x2?,x3?) = (tt(?a,x1,x2,x3)-tt(?a,x2,x1,x3)
                         +tt(?a,x2,x3,x1)-tt(?a,x3,x2,x1)
                         +tt(?a,x3,x1,x2)-tt(?a,x1,x3,x2))/6;
.sort
id  tt(x1?,x2?,x3?,x4?,x5?,x6?) = (tt(x1,x2,x3,x4,x5,x6)
                                  +tt(x4,x5,x6,x1,x2,x3))/2;
#call simd1d2
b   tt;
.sort
FillExpression T000000 = FT000000(tt);
```

```
Drop;
```

This tensor is considerably more complicated. The final statistics are

```
Time =      18.90 sec    Generated terms =      39816
       FT000000          Terms in output =      39816
                         Bytes used      =    4474128
```

and now we need to construct the tensors

```
FT000001  FT100000  FT100100  FT000011  FT110100
FT110000  FT111000  FT110110  FT111110  FT110111
FT111111  FT011111  FT111011  FT001111
```

These are all the T-tensors we need. The statistics are now at

```
Time =      98.00 sec    Generated terms =      55572
       FT001111          Terms in output =      55572
                         Bytes used      =    6030752
```

Now we are going to construct invariants. These are invariants that are formed by combinations of Weyl tensors and T-tensors with a total of four tensors. The simplest one is

```
Local Finv1 = Weyl0000(i1,i2,i3,i4)*
              Weyl1100(i1,i2,i5,i6)*
              Weyl1100(i3,i5,i7,i8)*
              Weyl1111(i4,i7,i6,i8);
#call contract8(Finv1,i1,i2,0,i3,i4,0,i5,i6,0,i7,i8)
```

Of course this needs an explanation of what the procedure contract8 looks like. It is not so complicated.

```
#procedure contract8(Expr,i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11)
#do j = 1,11
#if ( `i`j'' == 0 )
 id scqmu^2  = q*mu;
 id scqmu^-2 = 1/q/mu;
 .sort
#else
 sum `i`j'',1,...,10;
#endif
#enddo
id  scqmu^2  = q*mu;
id  scqmu^-2 = 1/q/mu;
.sort
#call simd1d2
id  d1 = d2^3-rho^2*L^2*mu;
id  rho^2 = d2-q;
Print +f +s;
.sort
```

```
Hide 'Expr';
#endprocedure
```

This procedure contracts the indexes in the order indicated and sorts when a parameter is zero. It calls the simplification routine and brings the notation to a unique form. Then it puts the expression in the hide system.

The result is

```
Time =      98.34 sec    Generated terms =          35
           Finv1         Terms in output =           9
                         Bytes used      =         436

   Finv1=
      +11823/8*d2^-12*mu^4*q^4
      -2652*d2^-11*mu^4*q^3
      +3168*d2^-10*mu^4*q^2
      -7144*d2^-9*L^-2*mu^3*q^3
      -1728*d2^-9*mu^4*q
      +6408*d2^-8*L^-2*mu^3*q^2
      +360*d2^-8*mu^4
      -1920*d2^-7*L^-2*mu^3*q
      +1664*d2^-6*L^-4*mu^2*q^2
      ;
```

When there are more contracted indexes we have corresponding procedures contract9 to contract12. A few of the other invariants are

```
    Local Finv3 = Weyl1111(i1,i2,i3,i4)*
                  Weyl0101(i1,i5,i6,i7)*
                  Weyl0111(i2,i6,i8,i9)*
                  T000000(i3,i4,i5,i7,i8,i9);
    #call contract9(Finv3,i3,i4,i5,0,i7,i8,i9,0,i1,i2,0,i6)
```

with

```
Time =     105.11 sec    Generated terms =          26
           Finv3         Terms in output =           7
                         Bytes used      =         412
```

```
   Finv3=
      +1332*d2^-9*L^-2*mu^3*q^3
      -364*d2^-9*L^-2*mu^3*q^3*u3^2
      -1216*d2^-8*L^-2*mu^3*q^2
      +256*d2^-8*L^-2*mu^3*q^2*u3^2
```

```
+320*d2^-7*L^-2*mu^3*q
-192*d2^-6*L^-4*mu^2*q^2
-320*d2^-6*L^-4*mu^2*q^2*u3^2
;
```

```
    Local Finv8 = Weyl0001(i1,i2,i3,i4)*
                  Weyl1011(i1,i5,i6,i7)*
                  T111111(i2,i3,i5,i8,i9,i10)*
                  T000000(i4,i6,i8,i7,i9,i10);
    #call contract10(Finv8,i4,i6,i7,0,i8,i9,i10,0,i1,i5,0,i2,i3)
```

with

```
Time =     139.87 sec    Generated terms =        105
           Finv8         Terms in output =         19
                         Bytes used      =        932
```

```
   Finv8=
      +461/6*d2^-12*mu^4*q^4
      +308/3*d2^-12*mu^4*q^4*u3^2
      -1/2*d2^-12*mu^4*q^4*u3^4
      -280*d2^-11*mu^4*q^3
      -408*d2^-11*mu^4*q^3*u3^2
      +288*d2^-10*mu^4*q^2
      +800/3*d2^-10*mu^4*q^2*u3^2
```

```
-26/3*d2^-9*L^-2*mu^3*q^3
+3812/3*d2^-9*L^-2*mu^3*q^3*u3^2
+1370/9*d2^-9*L^-2*mu^3*q^3*u3^4
-256/3*d2^-9*mu^4*q
-224/3*d2^-8*L^-2*mu^3*q^2
-4832/3*d2^-8*L^-2*mu^3*q^2*u3^2
-112*d2^-8*L^-2*mu^3*q^2*u3^4
-704/3*d2^-7*L^-2*mu^3*q
+192*d2^-7*L^-2*mu^3*q*u3^2
+1104*d2^-6*L^-4*mu^2*q^2
+1088*d2^-6*L^-4*mu^2*q^2*u3^2
+656/9*d2^-6*L^-4*mu^2*q^2*u3^4
;
```

and the worst one

```
Local Finv20 =T000000(i1,i2,i3,i4,i5,i6)*
              T100100(i1,i7,i8,i4,i9,i10)*
              T110110(i2,i7,i11,i5,i9,i12)*
              T111111(i3,i8,i11,i6,i10,i12);
#call contract12(Finv20,i1,i2,i3,0,i4,i5,i6,0,i7,i8,i9,i10,0,
                 i11,i12)
```

with

```
Time =    1332.71 sec    Generated terms =       1000
          Finv20         Terms in output =       1000
                         Bytes used      =     228896

Time =    1333.42 sec    Generated terms =      39816
          Finv20         Terms in output =      39816
                         Bytes used      =    5215504

Time =    1667.91 sec    Generated terms =   24660544
```

```
           Finv20          Terms in output =   12031168
                           Bytes used      = 1167649472

Time =     2340.05 sec     Generated terms =   11990644
           Finv20          Terms in output =       1802
                           Bytes used      =     108644

Time =     2340.09 sec     Generated terms =        173
           Finv20          Terms in output =         26
                           Bytes used      =       1004

   Finv20=
      +91406336*d2^-12*mu^4*q^4
      -28672000/3*d2^-12*mu^4*q^4*u3^2
      +272211968/3*d2^-12*mu^4*q^4*u3^4
      -688128*d2^-12*mu^4*q^4*u3^6
      +114688*d2^-12*mu^4*q^4*u3^8
      -547291136/3*d2^-11*mu^4*q^3
```

```
+47022080/3*d2^-11*mu^4*q^3*u3^2
-178913280*d2^-11*mu^4*q^3*u3^4
+688128*d2^-11*mu^4*q^3*u3^6
+273416192/3*d2^-10*mu^4*q^2
-18350080/3*d2^-10*mu^4*q^2*u3^2
+266076160/3*d2^-10*mu^4*q^2*u3^4
+592936960/3*d2^-9*L^-2*mu^3*q^3
+168132608/3*d2^-9*L^-2*mu^3*q^3*u3^2
+529399808/3*d2^-9*L^-2*mu^3*q^3*u3^4
+58032128/3*d2^-9*L^-2*mu^3*q^3*u3^6
-1605632/3*d2^-9*L^-2*mu^3*q^3*u3^8
-595460096/3*d2^-8*L^-2*mu^3*q^2
-60555264*d2^-8*L^-2*mu^3*q^2*u3^2
-177078272*d2^-8*L^-2*mu^3*q^2*u3^4
-64225280/3*d2^-8*L^-2*mu^3*q^2*u3^6
+413335552/3*d2^-6*L^-4*mu^2*q^2
+66060288*d2^-6*L^-4*mu^2*q^2*u3^2
+133955584*d2^-6*L^-4*mu^2*q^2*u3^4
```

```
+80740352/3*d2^-6*L^-4*mu^2*q^2*u3^6
+14221312/3*d2^-6*L^-4*mu^2*q^2*u3^8
;
```

This last one took about 1000 sec. The rest of the project was to add the 20 invariants with coefficients obtained by different means.

At the moment this program was constructed not all invariants were the 'correct ones' and not all coefficients were right. For the program that makes no difference. By the time all this was sorted out the program needed only minor modifications. The paper that contains the results of these calculations was put in the archive in Jan-2013 (arXiv:1301.6773 [hep-th]). The exact invariants needed and their coefficients can be looked up in that paper.