SetTensor, a package for *MathTensor*

S. Brandt

July 18, 1996

1 Introduction

MathTensor provides a series of powerful tools for evaluating tensor objects. These functions must be implemented in a specific sequence of steps if the desired output is to be obtained. First you must apply transformation rules, make sure derivative indices are lowered, evaluate covariant derivatives (the function `CDtoOD` may need to be called more than once to accomplish this and there is no builtin automatic way to repeat it), make sums, assign index values, and last of all to evaluate ordinary derivatives. For optimization purposes it is useful to have some quantities (like affine connections) pre-calculated. All this can be a bit of a nuissance to write out. The *SetTensor* package is designed to make this easier.

Some of the functionality of *SetTensor* can be obtained with the package *Components*, but there are a few advantages to using *SetTensor*. *Components* requires you to edit a template file to define the metric in the appropriate way, set flags, etc. No such file is needed with *SetTensor*. Also *Components* may calculate some quantities that you do not really need. You may only need the lower, and not the upper, indexed values of the Riemann tensor. With *SetTensor* you will only calculate the quantities that you need. *Components* does not allow you to change the size of your metric and the value of your metric components during a session, *SetTensor* does.

SetTensor adds new symbols to the environment, some of these are as follows: `EvalMT`, `Iter`, `SetSqrtDetg`, and `InitializeMetric`. In order to use these commands you need to first define some basic objects, most of which are standard to *MathTensor*. These standard objects are `Dimension` and the variables `x[1]`, `x[2]`, ... `x[Dimension]`. You may also want to define the value of `Sqrt[Abs[Detg]]`. Your next step is to call `InitializeMetric`.

`InitializeMetric` can be called in one of four ways:

```
MetricDown = { ... };
MetricUp = { ... }
(* will call Inverse[] to make MetricUp *)
InitializeMetric[MetricDown];
```

```
(* will call Inverse[] to make MetricUp, then apply MySimp *)
(* to simplify the resulting matrix and affine connections. *)
InitializeMetric[MetricDown,MySimp];
(* will simply accept the user's definition of MetricUp *)
InitializeMetric[MetricDown,MetricUp];
(* will simply accept the user's definition of MetricUp *)
(* and apply MySimp to the affine connection terms *)
InitializeMetric[MetricDown,MetricUp,MySimp];
```

Where the `MetricUp` and `MetricDown` are `Dimension`×`Dimension` matrices.

The last two of the methods above are espcially important. It may be that you are using a full 3-metric and do not wish to have to deal with the true metric inverse. You might prefer to deal with the metric in this form:

```
MetricDown = {
  {g11[x,y,z],g12[x,y,z],g13[x,y,z]},
  {g21[x,y,z],g22[x,y,z],g23[x,y,z]},
  {g31[x,y,z],g32[x,y,z],g33[x,y,z]}
};
MetricUp = {
  {gu11[x,y,z],gu12[x,y,z],gu13[x,y,z]},
  {gu21[x,y,z],gu22[x,y,z],gu23[x,y,z]},
  {gu31[x,y,z],gu32[x,y,z],gu33[x,y,z]}
};
```

This form for `MetricUp` will be much easier for Mathematica to handle than `Inverse[MetricDown]`.

For added convenience, a function `SubFun` is added to this package which can be used to replace a function (and all its derivatives) with a new form. The usage is illustrated by this example:

```
In[]=
    SubFun[f'[x]+f[x],f[x],x^2]
Out[]=
    2 x+x^2
```

2 The `EvalMT` command

Once these basic steps outlined above are accomplished, it is a straightforward matter to calculate the components of almost any tensor, or scalar object within the *MathTensor* framework. For a scalar, you need merely to issue the `EvalMT` command. A complete example follows:

```
<<SetTensor.m
x[1] = theta; x[2] = phi;
```

```
Dimension = 2;
InitializeMetric[{
  {f1[theta,phi],0},
  {0,f2[theta,phi]}}];
EvalMT[CD[foo[theta,phi],la,ua]]
```

The routine **EvalMT** will automatically lower all covariant derivative indicices, sum paired indices, and evaluate ordinary derivatives and affine connections.

If you wish to call **EvalMT** for an indexed quantity, you need to supply a second argument. It works this way:

```
EvalMT[RicciToAffine[RicciR[li,lj]],{li->-1,lj->-1}]
```

Note that because only the affine connections are defined, it is necessary for you to provide the explicit call to the *MathTensor* builtin command `RicciToAffine`. In evaluating tensors and components it is useful to remember the following additional *MathTensor* commands: `ScalarRtoAffine`, and `RiemannToAffine`.

3 SetSqrtDetg

In calculations involving the Levi-Civita symbol, `Epsilon`, one obtains (from *MathTensor* expressions like `Sqrt[Abs[Detg]]` or `Sqrt[Abs[Detg]]/Detg`. The `SetTensor` package provides a function called `SetSqrtDetg[]` which takes, as an argument, a function which is assumed to be the absolute value of the square root of the determinant.

`SetSqrtDetg[sign,val]` takes two arguments. The term `sign` must be a number with a value equal to either `1` or `-1`. `SetSqrtDetg[sign,val]` uses `val` to represent `Sqrt[Abs[Detg]]`, and `sign*val`^{2} to represent `Detg` when it does not appear otherwise.

4 Iter[Tensor Object,Code]

`Iter` will loop over the *unassigned* indices of a tensor object, taking into account the objects symmetries. The index variables applied to the tensor object are filled in for the *Code* section to use. For example:

```
DefineTensor[foo,{{2,1},1}];
Iter[foo[la,lb],
  Print[la," and ",lb];
];
```

Produces the following output:

```
PermWeight::sym: Symmetries of foo assigned
```

```
PermWeight::def: Object foo defined

-3 and -3
-3 and -2
-3 and -1
-2 and -2
-2 and -1
-1 and -1
```

If we wish `Iter` to avoid one or more of these indices we can use `Set1Component` which, as its name implies, sets one component. Running the code

```
Set1Component[foo[-1,-2],0];
Iter[foo[la,lb],
  Print[la," and ",lb];
];
```

produces

```
-3 and -3
-3 and -2
-3 and -1
-2 and -2
-1 and -1
```

Note that if a `foo` had been anti-symmetric in its two indices, then `foo[-1,-1]` would automatically get the value zero and `Iter` would not loop over that element. For example:

```
DefineTensor[foo,{{2,1},-1}];
Iter[foo[la,lb],
  Print[la," and ",lb];
];
```

Produces the following output:

```
PermWeight::sym: Symmetries of foo assigned

PermWeight::def: Object foo defined

-3 and -2
-3 and -1
-2 and -1
```

4.1 Assigning to Tensor Objects

You can use Iter to fill in the components of a tensor object. For example:

```
DefineTensor[foo,{{1},1}]
Iter[foo[la],
    Set1Component[
      foo[la],{f1[x],f2[x],f3[x]}[[-la]]
    ]
]
```

will assign the values `f1[x]`, etc. to the components of the tensor object `foo`. Note that `la` takes on negative values inside the `Iter` loop, so when the array is indexed we used a negative sign.

You can also combine this with `EvalMT` and evaluate the simplified components of the `Ricci` tensor.

```
DefineTensor[ric,{{2,1},1}];
Iter[ric[la,lb],
  Set1Component[
    ric[la,lb],Simplify[
      EvalMT[RicciToAffine[RicciR[lc,ld]],{lc->la,ld->lb}]
    ]
  ]
]
```

5 SetTensor

The package's namesake, the function `SetTensor`, provides a shorthand for calling `Iter`. The following examples illustrate how it works.

```
SetTensor[foo[la,ub],{
  {f1,f2,f3},
  {f4,f5,f6},
  {f7,f8,f9}}]
```

is the equivalent of

```
Iter[foo[la,ub],
  Set1Component[foo[la,ub],{
    {f1,f2,f3},
    {f4,f5,f6},
    {f7,f8,f9}}[[-la,ub]]
  ]
]
```

Notice that lower indices automatically get minus signs, but upper indices don't.

```
SetTensor[ric[la,lb],RicciToAffine[RicciR[la,lb]]]
```

is equivalent to

```
DefineTensor[ric,{{2,1},1}];
Iter[ric[la,lb],
  Set1Component[
    ric[la,lb],
      EvalMT[RicciToAffine[RicciR[lc,ld]],{lc->la,ld->lb}]
  ]
]
```

6 Naming Components of Arrays

It may be the case that, for the code you are writing, you wish to name all the components of an array in a similar way. Rather than typing out lengthy arrays, *SetTensor* provides a bit of shorthand for you.

6.1 AddDeps[Object], DefaultDepString

This is a very simple function which adds a dependency list to a string, then turns it into an expression. For example:

```
x[1] = x; x[2] = y; x[3] = z;
Dimension = 3;
AddDeps["psi"]
```

produces

```
Out[]=
  psi[x,y,z]
```

The utility of this function will become more apparent when it is combined with objects to be described in the next sections.

If you do not want the dependency list generated automatically from `x[1]`, `x[2]`, and `x[3]`, you can over-ride this by setting `DefaultDepString`. Continuing the above example we find that

```
DefaultDepString="x,y";
AddDeps["psi"]
```

produces

```
Out[]=
   psi[x,y];
```

.

6.2 AddIndex[Object]

`AddIndex` adds an index. To create a single vector named "U" you would do the following:

```
x[1] = x; x[2] = y; x[3] = z;
Dimension = 3;
AddIndex["U"];
```

And this would produce the output

```
Out[]=
  {"Ux","Uy","Uz"}
```

If we apply `AddDeps` to this we obtain

```
Out[]=
  {Ux[x, y, z], Uy[x, y, z], Uz[x, y, z]}
```

There is an optional argument to AddIndex, and its value can be either `Prepend` or `Append`. The latter is its default value. However, using the former value we obtain:

```
In[]=
  AddDeps[AddIndex[Prepend,"U"]]

Out[]=
  {xU[x, y, z], yU[x, y, z], zU[x, y, z]}
```

You can also apply `AddIndex` to itself.

```
In[]=
  AddDeps[AddIndex[AddIndex["U"]]]

Out[]=
{{Uxx[x, y, z], Uxy[x, y, z], Uxz[x, y, z]},
 {Uyx[x, y, z], Uyy[x, y, z], Uyz[x, y, z]},
 {Uzx[x, y, z], Uzy[x, y, z], Uzz[x, y, z]}}
```

6.3 AddSym2[Object], AddAsym2

The last example above showed how to create a 2 index tensor. What if we want that tensor to be symmetric? We need not really do anything, as the symmetries of the tensor cannot be violated by the assignment process using `SetTensor`. However, we might wind up with

```
In[]=
    foo[-2,-1]

Out[]=
    fooyx[x, y, z]
```

when we had wanted

```
Out[]=
    fooyx[x, y, z]
```

if we were not careful. For this purpose, and to provide a shorthand for adding 2 indices at once, we provide *AddSym2*.

```
In[]=
  AddDeps[AddSym2["U"]]

Out[]=
{{Uxx[x, y, z], Uxy[x, y, z], Uxz[x, y, z]},
 {Uxy[x, y, z], Uyy[x, y, z], Uyz[x, y, z]},
 {Uxz[x, y, z], Uyz[x, y, z], Uzz[x, y, z]}}
```

AddSym2 also takes the optional first argument which may be set to `Prepend` with effects similar to that for `AddIndex`. *AddAsym2* works the same way as *AddSym*, but obviously produces an anti-symmetric rather than a symmetric matrix.

```
In[]=
  AddDeps[AddAsym2["U"]]

Out[]=
{{0, Uxy[x, y, z], Uxz[x, y, z]},
 {-Uxy[x, y, z], 0, Uyz[x, y, z]},
 {-Uxz[x, y, z], -Uyz[x, y, z], 0}}
```

As a last example, let us make a matrix that has three indices, but is symmetric in the last two indices. The declaration looks like this:

```
tmp=AddIndex[AddSym2["U"]]
```

For this array, `tmp[[1,2,3]] == tmp[[1,3,2]]`.

7 Fortran Output

The package *Format*[1] with its function `FortranAssign` function replaces the builtin *Mathematica* function `FortranForm`. *Format* is automatically loaded

[1] For documentation, see http://www.wri.com/MathSource/.aliases/0205-254/Format.ps

by *SetTensor*. Note, however, that there is a namespace conflict between the two packages for the symbol `Lower`. Please ignore the warning message this generates during loading.

Several functions have been provided to interface between the package *Format* available from *MathSource* and *SetTensor*: `DerivRule` provides a mechanism for making the derivatives of functions format in a reasonable way, and `fwri` (Fortran write) simply calls `Write[]`, `DerivRule`, and `FortranAssign` with some specific default options.

7.1 DerivRule, MakeDeriv[String,String,String]

By default, `DerivRule` will convert a derivative to a function. Here are some examples to show you how it works:

```
In[]=
  D[foo[n,x],n] /. DerivRule

Out[]=
  dnfoo[n, x]

In[]=
  D[foo[n,x],x] /. DerivRule

Out[]=
  dxfoo[n, x]

In[]=
  D[foo[n,x],x,n] /. DerivRule

Out[]=
  dnxfoo[n, x]

In[]=
  D[foo[n,x],x,x] /. DerivRule

Out[]=
  dxxfoo[n, x]
```

The format of the derivative function is specified by `MakeDeriv`. MakeDeriv is called by `DerivRule` with three `String` arguments. The first specifies the variables we are using to take the derivatives, i.e. in the above examples we get “n”, “x”, “nx”, and “xx” respectively. The second argument specifies the name of the function, in the above example that would be “foo” in every case. The last argument is a string which supplies the function’s variable dependence, “[n,x]” in every example above.

The default definition of MakeDeriv is, therefore:

```
MakeDeriv[wr2_,fun_,args_] := ToExpression["d"<>wr2<>fun<>args]
```

You can over-ride this function to obtain the kind of expression you think is appropriate if the default is unacceptable.

7.2 fwri[Outputstream,lhs,rhs]

To use this function you need an Outputstream, i.e. the output of the Mathematica function OpenWrite for the first argument (for testing purposes you can use the string "stdout" to simply write to the screen). Next you supply the left-hand side and the right-hand side of the output you wish to produce. For example:

```
fwri["stdout",a,b]
```

produces (because fwri calls FortranAssign)

```
        a = b
```

appropriately spaced for punchcards and thus for modern Fortran compilers. Note that fwri automatically calles DerivRule, and provides a few mappings. The character sequence "UND" is mapped to the underscore character. The default list of arguments is mapped to the null string, "". To understand this, consider for example the spacetime specified by

```
x[1] = x; x[2] = y; x[3] = z;
Dimension = 3;
```

The default argument list for it is "(x,y,z)".

Now let us look at two slightly longer examples to see how all this works:

```
In[]=
  MakeDeriv[x1_,x2_,x3_] := ToExpression[
    x2<>"UND"<>x1<>x3];
  tmp=D[foo[n,x],x,x] /. DerivRule;
  fwri["stdout",a,tmp]

Out[]=
        a = foo_xx(n,x)

In[]=
  tmp=D[foo[x,y,z],x,x] /. DerivRule;
  fwri["stdout",a,tmp]

Out[]=
        a = foo_xx
```

In the second example we see that the argument list is suppressed. This is because it is the default list "(x,y,z)" given above.

If you call `fwri` with the *lhs* value set to the symbol `call`, then the assignment symbol "=" will be suppressed on output. Since `FortranAssign` converts integers to reals, we provide the symbol `FortranInt[x1_Integer]` to allow integers to format literally.

```
In[]=
  fwri["stdout",call,func[FortranInt[3]] ];

Out[]=
       call func(3)
```

If you are writing F90 code with array notation, this is what you want to have happen. However, it may be that you are writing F77 code and want "(x,y,z)" to map to "(i,j,k)" instead. If this is so, you can do the following:

```
In[]=
  FortranOutputOfDepList = "(i,j,k)";
  tmp=D[foo[x,y,z],x,x] /. DerivRule;
  fwri["stdout",a,tmp]

Out[]=
       a = foo_xx(i,j,k)
```

Note that you can supply all the arguments to `FortranAssign` that you might normally want to, however if you use `AssignReplace` (the mechanism used to produce the above string mappings) you lose the string mapping definitions supplied by SetTensor.

```
In[]=
  FortranOutputOfDepList = "(i,j,k)";
  tmp=D[foo[x,y,z],x,x] /. DerivRule;
  fwri["stdout",a,tmp,AssignRelplace->{"x"->"X"}];

Out[]=
        a = fooUNDXX(X,y,z)
```

7.3 AddReplace

This unfortunate state of affairs can be remedied by using the argument `AddAssign` as follows:

```
In[]=
  FortranOutputOfDepList = "(i,j,k)";
  tmp=D[foo[x,y,z],x,x] /. DerivRule;
```

```
  fwri["stdout",a,tmp,AddRelplace->{"x"->"X"}];

Out[]=
        a = foo_XX(X,y,z)
```

8 NeedsIt[],ClearNeedsIt[],AddToNeedsIt[]

Sometimes you may wish to only write out some components of a tensor. For example, suppose you want to calculate the Ricci scalar.

```
Needs["SetTensor`"];
Dimension=3;
x[1] = x; x[2] = y; x[3] = z;
DefaultDepString = "x,y";
gv = {
{AddDeps["f1"],0,0},
{0,AddDeps["f2"],0},
{0,0,AddDeps["f3"]}};
InitializeMetric[gv,Simplify];

ClearNeedsIt[];
fd = OpenWrite["rscal.h"];
rhs = ScalarRtoAffine[ScalarR];
rhs = rhs /. AffineG->af
rhs = EvalMT[rhs];
AddToNeedsIt[rhs];
fwri[fd,rsc,rhs];
Close[fd];
```

The function `NeedsIt` now knows which derivatives of the Affine connection you need. You can use this info to calculate only those components.

```
(* derivAf - Derivative of Affine Connection *)
DefineTensor[derivAf,{{1,3,2,4},1}];

(* the next two lines make the same affine symbol used above *)
tmp = AddIndex["A"];
tmp = AddSym2[tmp];

(* the next two lines prepend a dx, dy, or dz to the
 * affine connection name.
 *)
tmp = AddIndex[Prepend,tmp];
tmp = AddStrs["d",tmp];
```

```
tmp = AddDeps[tmp];
(* at this point tmp[[1,1,1,2]] yields dyAxxx[x,y] *)

rhs = OD[AffineG[ue,lf,lg],lh];
rhs = AffineToMetric[rhs];
rhs = Tsimplify[rhs];

fd = OpenWrite["dAf.h"];
Iter[derivAf[ua,lb,lc,ld],
  (* NeedsIt only returns True if the derivative
   * symbol was found in rhs in the call to
   * AddToNeedsIt above.
   *)
  If[NeedsIt[ tmp[[ua,-lb,-lc,-ld]] ],
    fwri[fd,tmp[[ua,-lb,-lc,-ld]],
      EvalMT[rhs,{ue->ua,lf->lb,lg->lc,lh->ld}]
    ]
  ]
]
Close[fd];
```

You can now include the files "dAf.h" and "rscal.h" in in your Fortran code, first "dAf.h" to set the Affine connection derivatives, then "rscal.h" which uses them.

Note that `AddToNeedsIt` applies `DerivRule` to its argument and then detects the functions with the default argument list. In the opinion of `AddToNeedsIt`, you only "need" functions that have the default argument list. (In this example, the default arguments are "x,y". This was explicitly set using `DefaultDepString` above. If it had not been set explicitly, the default arguments would've been "x,y,z".)

For reference, the file "dAf.h" contains the following:

```
      dxAxzz = 5.d-1*dxf1*dxf3/f1**2 - 5.d-1*dxxf3/f1
      dxAxyy = 5.d-1*dxf1*dxf2/f1**2 - 5.d-1*dxxf2/f1
      dyAxxy = -5.d-1*dyf1**2/f1**2 + 5.d-1*dyyf1/f1
      dyAyzz = 5.d-1*dyf2*dyf3/f2**2 - 5.d-1*dyyf3/f2
      dxAyxy = -5.d-1*dxf2**2/f2**2 + 5.d-1*dxxf2/f2
      dyAyxx = 5.d-1*dyf1*dyf2/f2**2 - 5.d-1*dyyf1/f2
      dyAzyz = -5.d-1*dyf3**2/f3**2 + 5.d-1*dyyf3/f3
      dxAzxz = -5.d-1*dxf3**2/f3**2 + 5.d-1*dxxf3/f3
```

and the file "rscal.h" contains:

```
      t1 = -(Axxy*Ayxx/f1) + Axxx*Ayxy/f1 - Ayxy**2/f1 + Ayxx*Ayyy/f1
```

```
     &  - Axxz*Azxx/f1 + Ayyz*Azxx/f1 - 2.d0*Ayxz*Azxy/f1 + Axxx*Azxz/f1
     &   - Azxz**2/f1 + Ayxx*Azyz/f1 + Azxx*Azzz/f1 - dxAyxy/f1 - dxAzxz
     &  /f1 + dyAyxx/f1 - Axxy**2/f2 + Axxx*Axyy/f2 - Axyy*Ayxy/f2 + Axx
     &  y*Ayyy/f2 - 2.d0*Axyz*Azxy/f2 + Axyy*Azxz/f2
        t2 = Axxz*Azyy/f2 - Ayyz*Azyy/f2 + Ayyy*Azyz/f2 - Azyz**2/f2 + A
     &  zyy*Azzz/f2 + dxAxyy/f2 - dyAxxy/f2 - dyAzyz/f2 - Axxz**2/f3 + A
     &  xxx*Axzz/f3 + Axzz*Ayxy/f3 - 2.d0*Axyz*Ayxz/f3 - Ayyz**2/f3 + Ax
     &  xy*Ayzz/f3 + Ayyy*Ayzz/f3 - Axzz*Azxz/f3 - Ayzz*Azyz/f3 + Axxz*A
     &  zzz/f3 + Ayyz*Azzz/f3 + dxAxzz/f3 + dyAyzz/f3
        rsc = t1 + t2
```

9 Optimization

It is always desirable to reduce the number of quantities you need to calculate, and the number of indices you need to contract. Some of this has been taken account for you, by assuming that you always want the affine connections. The command

```
EvalMT[RicciRVal[li,lj],AffineToMetric[RicciToAffine[RicciR[li,lj]]] ];
```

will take much longer to evaluate than

```
EvalMT[RicciRVal[li,lj],RicciToAffine[RicciR[li,lj]] ];
```

simply because the affine connections have already been calculated and one level of summation has been removed. If the affine connections have been simplified, this helps speed things up all the more.

When calculating the Riemann invariants, for another example, it is much more efficient to evaluate this

```
sym=Symmetries[RiemannR[la,lb,lc,ld]];
DefineTensor[riem,"r",sym];
SetTensor[riem[ua,ub,lc,ld],RiemannToAffine[RiemannR[ua,ub,lc,ld]]];
EvalMT[riem[ua,ub,lc,ld] riem[uc,ud,la,lb]]
```

than it is to do this

```
sym=Symmetries[RiemannR[la,lb,lc,ld]];
DefineTensor[riem,"r",sym];
SetTensor[riem[ua,ub,uc,ud],RiemannToAffine[RiemannR[ua,ub,uc,ud]]];
SetTensor[riem[la,lb,lc,ld],RiemannToAffine[RiemannR[la,lb,lc,ld]]];
EvalMT[riem[ua,ub,uc,ud] riem[lc,ld,la,lb]]
```

It also helps to apply `Simplify` to the components of `RiemannR`.

10 Timings

```
Needs["SetTensor`"];
(* the next file defines g00,g11,g22,g33,b3 *)
<<../BasicKerr.m;

Dimension = 4;
(* q is theta, p is phi *)
x[2] = r; x[3] = q; x[4] = p; x[1] = t;
metdn = {
  {g00,0,0,b3},
  {0,g11,0,0},
  {0,0,g22,0},
  {b3,0,0,g33}};
dtg = g00 g33-b3\^{}2;
metup = {
  {g33/dtg,0,0,-b3/dtg},
  {0,1/g11,0,0},
  {0,0,1/g22,0},
  {-b3/dtg,0,0,g00/dtg}};

On[EvalMT::PrintIndex];

sym=Symmetries[RiemannR[la,lb,lc,ld];
DefineTensor[riem,"r",sym];

Timing[
  InitializeMetric[metdn,metup,Simplify];
  SetTensor[riem[ua,ub,lc,ld],
    RiemannToAffine[RiemannR[ua,ub,lc,ld]],Simplify];
  Print["EvalMT..."];
  tmp=EvalMT[riem[ua,ub,lc,ld] riem[uc,ud,la,lb]];
  stmp=Simplify[tmp];
]
```

The above calculation took 3341.04 seconds on jean-luc, an RS6000. Making the substitution $u = a\ \cos\theta$ to remove the trigonmetric functions reduced the time to 549.81 seconds. By substituting `FactorSquareFree` for `Simplify` the time was reduced to 285.84 seconds. Setting the variable m to unity (which merely involves a rescaling of r and a), because it reduces the number of variables the simplification routines need to deal with, further reduced the run time of this calculation to 251.05 seconds.

11 Acknowledgements, Etc.

Much credit is due to Peter Musgrave (actually, almost everything I say in the optimization section I learned from him). I also want to acknowledge helpful conversations with the people of Wolfram Research. Unfortunately, this package is still an order of magnitude *slower* than the equivalent packages in *grtensor* as composed by Peter. I cannot imagine how he does it.