$\mathcal{GA}$lgebra: a Geometric Algebra Module for *Sympy*

Alan Bromborsky
Army Research Lab (Retired)
abrombo@verizon.net

May 21, 2015

Introduction

This document describes the implementation, installation and use of a geometric algebra module written in python that utilizes the *sympy* symbolic algebra library. The python module ga has been developed for coordinate free calculations using the operations (geometric, outer, and inner products etc.) of geometric algebra. The operations can be defined using a completely arbitrary metric defined by the inner products of a set of arbitrary vectors or the metric can be restricted to enforce orthogonality and signature constraints on the set of vectors. Additionally, a metric that is a function of a coordinate set can be defined so that a geometric algebra over a manifold can be implemented. Geometric algebras over submanifolds of the base manifold are also supported as well as linear multivector differential operators and linear transformations. In addition the module includes the geometric, outer (curl) and inner (div) derivatives. The module requires the *sympy* module and the numpy module for numerical linear algebra calculations. For LaTeXoutput a LaTeXdistribution must be installed.

Contents

Chapter 1

Installation on Linux, Windows, and Mac

1.1 Install python

The `galgebra` python module, which is an implementation of geometric algebra in python has two perquisites for a minimal installation, python and `sympy`. For the python language we have the following situation[1].

os	python installation
linux	Comes with all versions of linux
windows	To install python on windows go to https://www.python.org/downloads/windows/ and install version appropriate for you version of windows. If you wish a more complete/advanced installation go to https://code.google.com/p/pythonxy/.
mac	Basic version comes with OSX. For better installation go to http://docs.python-guide.org/en/latest/starting/install/osx/.

[1] Currently `galgebra` supports python versions 2.7+, but not versions 3.0+ of python.

1.2 Install sympy

For `sympy` there are two alternatives for installation.

mode	method
latest release	Go to https://github.com/sympy/sympy/releases and select option appropriate for your system.
development version	Go to https://github.com/sympy/sympy and download zipped archive. Unzip archive. Open terminal/command line in top directory of unzipped archive. For linux or osx run "`sudo python setup.py install`." For windows run "`python setup.py install`" from the command line.

The method for the development version is preferred since that method always builds `sympy` with the python system you have installed on your system (32-bits verses 64-bits and particular python version you are running).

1.3 Install galgebra

Since you are reading this document you have already obtained a copy of `galgebra`. If you wish to obtain the very latest version (assuming you have not already done this) go to https://github.com/brombo/galgebra and download and extract the zipped archive.

Then with whatever version you are using open a terminal/command line in the `galgebra` directory that is in the top directory of the archive. If you are in the correct the directory it should contain the python program `setgapth.py`. If you are in linux or osx run the program with the command `sudo python setgapth.py`, if in windows use `python setgapth.py`.

This program creates the file `Ga.pth` in the correct directory to simplify importing the `galgebra` modules into your python programs. The modules you will use for programming with geometric algebra/calculus are `ga`, `mv`, `lt`, and `printer`[2]. To import any of these modules into your program, say `mv`, you only have to enter in the program `import mv`. It does not matter where the program file is located.

[2] All these modules are in the same directory as `setgapth.py`.

1.4 LaTeX Options

In order to use the latex output of the `galgebra` modules (excluding latex output from *Ipython notebook*) you must install a latex distribution. Directions follow if you do not already have LaTeX installed.[3]

os	latex installation
linux	Open a terminal and run "`sudo apt-get texlive-full install`". It takes about half an hour to install.
windows	Go to `http://miktex.org/download` (other downloads). Download a net installer. Install a full version of *MikTex*.
mac	Go to `http://www.tug.org/mactex/` and follow instructions to install *MacTeX*.

1.5 "Ipython notebook" Options

To use "ipython notebook" with `galgebra` it must be installed. To install "ipython notebook" do the following.

Go to `http://pip.readthedocs.org/en/latest/installing.html` and install `pip`. Then run in a terminal/command line "`pip install "ipython[notebook]"`".

1.6 The ANSI Console

The `printer` module of `galgebra` contains the class `Eprint` which is described in section (3.12). This function uses the capabilities of the ansi console (terminal) for enhanced multivector printing where multivector bases, sympy functions and derivatives are printed in different colors. The ansi console is native to Linux and OSX, but not windows. The best available free substitute for the ansi console on windows is *ConEmu*. The web page for ConEmu is `http://conemu.github.io/`. In order to install *ConEmu* download the appropriate version of the *ConEmu* installer (exe file) for your system (32 bit or 64 bit) from the website and and execute it. Instructions for using *ConEmu* are given in section (3.12).

[3] In order for `galgebra` to output latex formatted pdf files your distribution of latex must have `pdflatex` installed.

1.7 Geany Programmers Editor

Geany is a very nice *free* programmers editor that work well with *python*. From within *geany* you can execute a *python* program. The *galgebra* printing system is setup so that you can display the program output on an ansi terminal or if you are using the LaTeXoptions has the terminal launch a *pdf* browser to view the LaTeXoutput. To install *geany* on Linux use the command line "`sudo apt-get install geany`", on Windows go to http://www.geany.org/Download/Releases or to install *geany* in OSX go to http://wiki.geany.org/howtos/osx/running.

Chapter 2

What is Geometric Algebra?

Geometric algebra is the Clifford algebra of a real finite dimensional vector space or the algebra that results when the vector space is extended with a product of vectors (geometric product) that is associative, left and right distributive, and yields a real number for the square (geometric product) of any vector [2], [1]. The elements of the geometric algebra are called multivectors and consist of the linear combination of scalars, vectors, and the geometric product of two or more vectors. The additional axioms for the geometric algebra are that for any vectors a, b, and c in the base vector space ([1],p85):

$$\begin{gathered} a\left(bc\right) = \left(ab\right)c \\ a\left(b+c\right) = ab + ac \\ \left(a+b\right)c = ac + bc \\ aa = a^2 \in \Re \end{gathered} \tag{2.1}$$

The dot product of two vectors is defined by ([1],p86)

$$a \cdot b \equiv \left(ab + ba\right)/2 \tag{2.2}$$

Then consider

$$c = a + b \tag{2.3}$$

$$c^2 = \left(a+b\right)^2 \tag{2.4}$$

$$c^2 = a^2 + ab + ba + b^2 \tag{2.5}$$
$$a \cdot b = (c^2 - a^2 - b^2)/2 \in \Re \tag{2.6}$$

Thus $a \cdot b$ is real. The objects generated from linear combinations of the geometric products of vectors are called multivectors. If a basis for the underlying vector space is the set of vectors formed from $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n$ (we use boldface $\boldsymbol{e}$'s to denote basis vectors) a complete basis for the geometric algebra is given by the scalar 1, the vectors $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n$ and all geometric products of vectors

$$\boldsymbol{e}_{i_1}\boldsymbol{e}_{i_2}\ldots\boldsymbol{e}_{i_r} \text{ where } 0 \le r \le n,\ 0 \le i_j \le n \text{ and } i_1 < i_2 < \cdots < i_r \tag{2.7}$$

Each base of the complete basis is represented by a non-commutative symbol (except for the scalar 1) with name $\boldsymbol{e}_{i_1}\ldots\boldsymbol{e}_{i_r}$ so that the general multivector $\boldsymbol{A}$ is represented by (A is the scalar part of the multivector and the $A^{i_1,\ldots,i_r}$ are scalars)

$$\boldsymbol{A} = A + \sum_{r=1}^{n} \sum_{\substack{i_1,\ldots,i_r \\ 0 \le i_j < i_{j+1} \le n}} A^{i_1,\ldots,i_r}\boldsymbol{e}_{i_1}\boldsymbol{e}_{i_2}\ldots\boldsymbol{e}_r \tag{2.8}$$

The critical operation in setting up the geometric algebra is reducing the geometric product of any two bases to a linear combination of bases so that we can calculate a multiplication table for the bases. Since the geometric product is associative we can use the operation (by definition for two vectors $a \cdot b \equiv (ab + ba)/2$ which is a scalar)

$$\boldsymbol{e}_{i_{j+1}}\boldsymbol{e}_{i_j} = 2\boldsymbol{e}_{i_{j+1}} \cdot \boldsymbol{e}_{i_j} - \boldsymbol{e}_{i_j}\boldsymbol{e}_{i_{j+1}} \tag{2.9}$$

These processes are repeated until every basis list in $\boldsymbol{A}$ is in normal (ascending) order with no repeated elements. As an example consider the following

$$\begin{aligned}
\boldsymbol{e}_3\boldsymbol{e}_2\boldsymbol{e}_1 &= (2(\boldsymbol{e}_2 \cdot \boldsymbol{e}_3) - \boldsymbol{e}_2\boldsymbol{e}_3)\boldsymbol{e}_1 && (2.10)\\
&= 2\,(\boldsymbol{e}_2 \cdot \boldsymbol{e}_3)\,\boldsymbol{e}_1 - \boldsymbol{e}_2\boldsymbol{e}_3\boldsymbol{e}_1 && (2.11)\\
&= 2\,(\boldsymbol{e}_2 \cdot \boldsymbol{e}_3)\,\boldsymbol{e}_1 - \boldsymbol{e}_2\,(2\,(\boldsymbol{e}_1 \cdot \boldsymbol{e}_3) - \boldsymbol{e}_1\boldsymbol{e}_3) && (2.12)\\
&= 2\,((\boldsymbol{e}_2 \cdot \boldsymbol{e}_3)\,\boldsymbol{e}_1 - (\boldsymbol{e}_1 \cdot \boldsymbol{e}_3)\,\boldsymbol{e}_2) + \boldsymbol{e}_2\boldsymbol{e}_1\boldsymbol{e}_3 && (2.13)
\end{aligned}$$

$$= 2\left(\left(\boldsymbol{e}_2 \cdot \boldsymbol{e}_3\right)\boldsymbol{e}_1 - \left(\boldsymbol{e}_1 \cdot \boldsymbol{e}_3\right)\boldsymbol{e}_2 + \left(\boldsymbol{e}_1 \cdot \boldsymbol{e}_2\right)\boldsymbol{e}_3\right) - \boldsymbol{e}_1\boldsymbol{e}_2\boldsymbol{e}_3 \tag{2.14}$$

which results from repeated application of eq. (2.9). If the product of basis vectors contains repeated factors eq. (2.9) can be used to bring the repeated factors next to one another so that if $\boldsymbol{e}_{i_j} = \boldsymbol{e}_{i_{j+1}}$ then $\boldsymbol{e}_{i_j}\boldsymbol{e}_{i_{j+1}} = \boldsymbol{e}_{i_j} \cdot \boldsymbol{e}_{i_{j+1}}$ which is a scalar that commutes with all the terms in the product and can be brought to the front of the product. Since every repeated pair of vectors in a geometric product of r factors reduces the number of non-commutative factors in the product by $r-2$. The number of bases in the multivector algebra is 2^n and the number containing r factors is $\binom{n}{r}$ which is the number of combinations or n things taken r at a time (binomial coefficient).

The other construction required for formulating the geometric algebra is the outer or wedge product (symbol $\wedge$) of r vectors denoted by $a_1 \wedge \ldots \wedge a_r$. The wedge product of r vectors is called an r-blade and is defined by ([1],p86)

$$a_1 \wedge \ldots \wedge a_r \equiv \sum_{i_{j_1} \ldots i_{j_r}} \epsilon^{i_{j_1} \ldots i_{j_r}} a_{i_{j_1}} \ldots a_{i_{j_1}} \tag{2.15}$$

where $\epsilon^{i_{j_1} \ldots i_{j_r}}$ is the contravariant permutation symbol which is $+1$ for an even permutation of the superscripts, 0 if any superscripts are repeated, and -1 for an odd permutation of the superscripts. From the definition $a_1 \wedge \ldots \wedge a_r$ is antisymmetric in all its arguments and the following relation for the wedge product of a vector a and an r-blade B_r can be derived

$$a \wedge B_r = (aB_r + (-1)^r B_r a)/2 \tag{2.16}$$

Using eq. (2.16) one can represent the wedge product of all the basis vectors in terms of the geometric product of all the basis vectors so that one can solve (the system of equations is lower diagonal) for the geometric product of all the basis vectors in terms of the wedge product of all the basis vectors. Thus a general multivector $\boldsymbol{B}$ can be represented as a linear combination of a scalar and the basis blades.

$$\boldsymbol{B} = B + \sum_{r=1}^{n} \sum_{i_1,\ldots,i_r,\ \forall\ 0 \le i_j \le n} B^{i_1,\ldots,i_r} \boldsymbol{e}_{i_1} \wedge \boldsymbol{e}_{i_2} \wedge \ldots \wedge \boldsymbol{e}_r \tag{2.17}$$

Using the blades $\boldsymbol{e}_{i_1} \wedge \boldsymbol{e}_{i_2} \wedge \ldots \wedge \boldsymbol{e}_r$ creates a graded algebra where r is the grade of the basis blades. The grade-r part of $\boldsymbol{B}$ is the linear combination of all terms with grade r basis blades.

The scalar part of $\boldsymbol{B}$ is defined to be grade-0. Now that the blade expansion of $\boldsymbol{B}$ is defined we can also define the grade projection operator $\langle \boldsymbol{B} \rangle_r$ by

$$\langle \boldsymbol{B} \rangle_r = \sum_{i_1,\ldots,i_r,\ \forall\ 0 \le i_j \le n} B^{i_1,\ldots,i_r} \boldsymbol{e}_{i_1} \wedge \boldsymbol{e}_{i_2} \wedge \ldots \wedge \boldsymbol{e}_r \tag{2.18}$$

and

$$\langle \boldsymbol{B} \rangle \equiv \langle \boldsymbol{B} \rangle_0 = B \tag{2.19}$$

Then if $\boldsymbol{A}_r$ is an r-grade multivector and $\boldsymbol{B}_s$ is an s-grade multivector we have

$$\boldsymbol{A}_r \boldsymbol{B}_s = \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{|r-s|} + \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{|r-s|+2} + \cdots \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{r+s} \tag{2.20}$$

and define ([2],p6)

$$\boldsymbol{A}_r \wedge \boldsymbol{B}_s \equiv \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{r+s} \tag{2.21}$$

$$\boldsymbol{A}_r \cdot \boldsymbol{B}_s \equiv \left\{ \begin{array}{cc} r \text{ and } s \neq 0: & \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{|r-s|} \\ r \text{ or } s = 0: & 0 \end{array} \right\} \tag{2.22}$$

where $\boldsymbol{A}_r \cdot \boldsymbol{B}_s$ is called the dot or inner product of two pure grade multivectors. For the case of two non-pure grade multivectors

$$\boldsymbol{A} \wedge \boldsymbol{B} = \sum_{r,s} \langle \boldsymbol{A} \rangle_r \wedge \langle \boldsymbol{B} \rangle_s \tag{2.23}$$

$$\boldsymbol{A} \cdot \boldsymbol{B} = \sum_{r,s \neq 0} \langle \boldsymbol{A} \rangle_r \cdot \langle \boldsymbol{B} \rangle_s \tag{2.24}$$

Two other products, the left ($\rfloor$) and right ($\lfloor$) contractions, are defined by

$$\boldsymbol{A} \lfloor \boldsymbol{B} \equiv \sum_{r,s} \left\{ \begin{array}{cc} \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{r-s} & r \geq s \\ 0 & r < s \end{array} \right\} \tag{2.25}$$

$$\boldsymbol{A}\rfloor\boldsymbol{B} \equiv \sum_{r,s} \left\{ \begin{array}{cc} \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{s-r} & s \geq r \\ 0 & s < r \end{array} \right\} \tag{2.26}$$

A final operation for multivectors is the reverse. If a multivector $\boldsymbol{A}$ is the geometric product of r vectors (versor) so that $\boldsymbol{A} = a_1 \ldots a_r$ the reverse is defined by

$$\boldsymbol{A}^\dagger \equiv a_r \ldots a_1 \tag{2.27}$$

where for a general multivector we have (the the sum of the reverse of versors)

$$\boldsymbol{A}^\dagger = A + \sum_{r=1}^{n} (-1)^{r(r-1)/2} \sum_{i_1,\ldots,i_r,\, \forall\, 0 \leq i_j \leq n} A^{i_1,\ldots,i_r} \boldsymbol{e}_{i_1} \wedge \boldsymbol{e}_{i_2} \wedge \ldots \wedge \boldsymbol{e}_r \tag{2.28}$$

note that if $\boldsymbol{A}$ is a versor then $\boldsymbol{A}\boldsymbol{A}^\dagger \in \Re$ and $(AA^\dagger \neq 0)$

$$\boldsymbol{A}^{-1} = \frac{\boldsymbol{A}^\dagger}{\boldsymbol{A}\boldsymbol{A}^\dagger} \tag{2.29}$$

2.1 Representation of Multivectors in *sympy*

The *sympy* python module offers a simple way of representing multivectors using linear combinations of commutative expressions (expressions consisting only of commuting *sympy* objects) and non-commutative symbols. We start by defining n non-commutative *sympy* symbols as a basis for the vector space

```
(e_1,...,e_n) = symbols('e_1,...,e_n',commutative=False,real=True)
```

Several software packages for numerical geometric algebra calculations are available from Doran-Lasenby group and the Dorst group. Symbolic packages for Clifford algebra using orthogonal bases such as $\boldsymbol{e}_i\boldsymbol{e}_j + \boldsymbol{e}_j\boldsymbol{e}_i = 2\eta_{ij}$, where η_{ij} is a numeric array are available in Maple and Mathematica. The symbolic algebra module, *ga*, developed for python does not depend on an orthogonal basis representation, but rather is generated from a set of n arbitrary symbolic vectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$ and a symbolic metric tensor $g_{ij} = \boldsymbol{e}_i \cdot \boldsymbol{e}_j$ (the symbolic metric can be symbolic constants or symbolic function in the case of a manifold).

$A + B$	sum of multivectors
$A - B$	difference of multivectors
$A * B$	geometric product of multivectors
$A \wedge B$	outer product of multivectors
$A \vert B$	inner product of multivectors
$A < B$	left contraction of multivectors
$A > B$	right contraction of multivectors

Table 2.1: Multivector operations for GA

In order not to reinvent the wheel all scalar symbolic algebra is handled by the python module *sympy* and the abstract basis vectors are encoded as non-commuting *sympy* symbols.

The basic geometric algebra operations will be implemented in python by defining a geometric algebra class, *Ga*, that performs all required geometric algebra an calculus operations on *sympy* expressions of the form (Einstein summation convention)

$$F + \sum_{r=1}^{n} F^{i_1 \cdots i_r} \boldsymbol{e}_{i_1} \ldots \boldsymbol{e}_{i_r} \tag{2.30}$$

where the F's are *sympy* symbolic constants or functions of the coordinates and a multivector class, *Mv*, that wraps *Ga* and overloads the python operators to provide all the needed multivector operations as shown in Table 2.1 where A and B are any two multivectors (In the case of $+$, $-$, $*$, $\wedge$, $|$, $<$, and $>$ the operation is also defined if A or B is a *sympy* symbol or a *sympy* real number).

Since `<` and `>` have no r-forms (in python for the `<` and `>` operators there are no `__rlt__()` and `__rgt__()` member functions to overload) we can only have mixed modes (sympy scalars and multivectors) if the first operand is a multivector.

Except for `<` and `>` all the multivector operators have r-forms so that as long as one of the operands, left or right, is a multivector the other can be a multivector or a scalar (*sympy* symbol or integer).

Note that the operator order precedence is determined by python and is not necessarily that used by geometric algebra. It is *absolutely essential* to use parenthesis in multivector expressions containing `^`, `|`, `<`, and/or `>`. As an example let `A` and `B` be any two multivectors. Then `A + A*B = A +(A*B)`, but `A+A^B = (2*A)^B` since in python the `^` operator has a lower precedence than the `+` operator. In geometric algebra the outer and inner products and the left and right contractions have a higher precedence than the geometric product and the geometric product

has a higher precedence than addition and subtraction. In python the ^, |, >, and < all have a lower precedence than + and - while * has a higher precedence than + and -.

For those users who wish to define a default operator precedence the functions def_prec() and GAeval() are available in the module printer.

def_prec(gd,op_ord='<>|,^,*')

> Define the precedence of the multivector operations. The function def_prec() must be called from the main program and the first argument gd must be set to globals(). The second argument op_ord determines the operator precedence for expressions input to the function GAeval(). The default value of op_ord is '<>|,^,*'. For the default value the <, >, and | operations have equal precedence followed by ^, and ^ is followed by *.

GAeval(s,pstr=False)

> The function GAeval() returns a multivector expression defined by the string s where the operations in the string are parsed according to the precedences defined by define_precedence(). pstr is a flag to print the input and output of GAeval() for debugging purposes. GAeval() works by adding parenthesis to the input string s with the precedence defined by op_ord='<>|,^,*'. Then the parsed string is converted to a *sympy* expression using the python eval() function. For example consider where X, Y, Z, and W are multivectors
>
> ```
> def_prec(globals())
> V = GAeval('X|Y^Z*W')
> ```
>
> The *sympy* variable V would evaluate to ((X|Y)^Z)*W.

2.2 Vector Basis and Metric

The two structures that define the metric class (inherited by the geometric algebra class) are the symbolic basis vectors and the symbolic metric. The symbolic basis vectors are input as a string with the symbol name separated by spaces. For example if we are calculating the geometric algebra of a system with three vectors that we wish to denote as a0, a1, and a2 we would define the string variable:

```
basis = 'a0 a1 a2'
```

that would be input into the geometric algebra class instantiation function, `Ga()`. The next step would be to define the symbolic metric for the geometric algebra of the basis we have defined. The default metric is the most general and is the matrix of the following symbols

$$g = \begin{bmatrix} (a0.a0) & (a0.a1) & (a0.a2) \\ (a0.a1) & (a1.a1) & (a1.a2) \\ (a0.a2) & (a1.a2) & (a2.a2) \end{bmatrix} \tag{2.31}$$

where each of the g_{ij} is a symbol representing all of the dot products of the basis vectors. Note that the symbols are named so that $g_{ij} = g_{ji}$ since for the symbol function $(a0.a1) \neq (a1.a0)$.

Note that the strings shown in eq. (2.31) are only used when the values of g_{ij} are output (printed). In the ga module (library) the g_{ij} symbols are stored in a member of the geometric algebra instance so that if `o3d` is a geometric algebra then `o3d.g` is the metric tensor (g_{ij} =`o3d.g[i,j]`) for that algebra.

The default definition of g can be overwritten by specifying a string that will define g. As an example consider a symbolic representation for conformal geometry. Define for a basis

```
basis = 'a0 a1 a2 n nbar'
```

and for a metric

```
g = '# # # 0 0, # # # 0 0, # # # 0 0, 0 0 0 0 2, 0 0 0 2 0'
```

then calling `cf3d = Ga(basis,g=g)` would initialize the metric tensor

$$g = \begin{bmatrix} (a0.a0) & (a0.a1) & (a0.a2) & 0 & 0 \\ (a0.a1) & (a1.a1) & (a1.a2) & 0 & 0 \\ (a0.a2) & (a1.a2) & (a2.a2) & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix} \tag{2.32}$$

for the `cf3d` (conformal 3-d) geometric algebra.

Here we have specified that `n` and `nbar` are orthogonal to all the `a`'s, `(n.n) = (nbar.nbar) = 0`, and `(n.nbar) = 2`. Using `#` in the metric definition string just tells the program to use the default symbol for that value.

When `Ga` is called multivector representations of the basis local to the program are instantiated. For the case of an orthogonal 3-d vector space that means the symbolic vectors named `a0`, `a1`, and `a2` are created. We can instantiate the geometric algebra and obtain the basis vectors with -

```
o3d = Ga('a_1 a_2 a_3',g=[1,1,1])
(a_1,a_2,a_3) = o3d.mv()
```

or use the `Ga.build()` function -

```
(o3d,a_1,a_2,a_3) = Ga.build('a_1 a_2 a_3',g=[1,1,1])
```

Note that the python variable name for a basis vector does not have to correspond to the name give in `Ga()` or `Ga.build()`, one may wish to use a shortened python variable name to reduce programming (typing) errors, for example one could use -

```
(o3d,a1,a2,a3) = Ga.build('a_1 a_2 a_3',g=[1,1,1])
```

or

```
(st4d,g0,g1,g2,g3) = Ga.build('gamma_0 gamma_1 gamma_2 gamma_3',\
                              g=[1,-1,-1,-1])
```

for Minkowski space time.

If the latex printer is used `e1` would print as $\boldsymbol{e_1}$ and `g1` as $\boldsymbol{\gamma_1}$.

Additionally `Ga()` and `Ga.build()` has simplified options for naming a set of basis vectors and for inputing an orthogonal basis. If one wishes to name the basis vectors $\boldsymbol{e}_x$, $\boldsymbol{e}_y$, and $\boldsymbol{e}_z$ then set `basis='e*x|y|z'` or to name $\boldsymbol{\gamma}_t$, $\boldsymbol{\gamma}_x$, $\boldsymbol{\gamma}_y$, and $\boldsymbol{\gamma}_z$ then set `basis='gamma*t|x|y|z'`. For the case of an orthogonal basis if the signature of the vector space is $(1,1,1)$ (Euclidean 3-space) set `g=[1,1,1]` or if it is $(1,-1,-1,-1)$ (Minkowski 4-space) set `g=[1,-1,-1,-1]`. If `g` is a function of position then `g` can be entered as a *sympy* matrix with *sympy* functions as the entries of the matrix or as a list of functions for the case of a orthogonal metric. In the case of spherical coordinates we have `g=[1,r**2,r**2*sin(th)**2]`.

2.3 Representation and Reduction of Multivector Bases

In our symbolic geometric algebra all multivectors can be obtained from the symbolic basis vectors we have input, via the different operations available to geometric algebra. The first problem we have is representing the general multivector in terms terms of the basis vectors. To do this we form the ordered geometric products of the basis vectors and develop an internal representation of these products in terms of python classes. The ordered geometric products are all multivectors of the form $a_{i_1}a_{i_2}\dots a_{i_r}$ where $i_1 < i_2 < \cdots < i_r$ and $r \leq n$. We call these multivectors bases and represent them internally with non-commutative symbols so for example $a_1a_2a_3$ is represented by

```
Symbol('a_1*a_2*a_3',commutative=False)
```

In the simplest case of two basis vectors `a_1` and `a_2` we have a list of bases

```
self.bases = [[Symbol('a_1',commutative=False,real=True),\
              Symbol('a_2',commutative=False,real=True)],\
              [Symbol('a_1*a_2',commutative=False,real=True)]]
```

For the case of the basis blades we have

```
self.blades = [[Symbol('a_1',commutative=False,real=True),\
               Symbol('a_2',commutative=False,real=True)],\
               [Symbol('a_1^a_2',commutative=False,real=True)]]
```

> For all grades/pseudo-grades greater than one (vectors) the `*` in the name of the base symbol is replaced with a `^` in the name of the blade symbol so that for all basis bases and blades of grade/pseudo-grade greater than one there are different symbols for the corresponding bases and blades.

The index tuples for the bases of each pseudo grade and each grade for the case of dimension 3 is

```
self.indexes = (((0,),(1,),(2,)),((0,1),(0,2),(1,2)),((0,1,2)))
```

Then the non-commutative symbol representing each base is constructed from each index tuple. For example for `self.indexes[1][1]` the symbol is `Symbol('a_1*a_3',commutative=False)`.

> In the case that the metric tensor is diagonal (orthogonal basis vectors) both base and blade bases are identical and fewer arrays and dictionaries need to be constructed.

2.4 Base Representation of Multivectors

In terms of the bases defined as non-commutative *sympy* symbols the general multivector is a linear combination (scalar *sympy* coefficients) of bases so that for the case of two bases the most general multivector is given by -

```
A = A_0+A__1*self.bases[1][0]+A__2*self.bases[1][1]+\
    A__12*self.bases[2][0]
```

If we have another multivector `B` to multiply with `A` we can calculate the product in terms of a linear combination of bases if we have a multiplication table for the bases.

2.5 Blade Representation of Multivectors

Since we can now calculate the symbolic geometric product of any two multivectors we can also calculate the blades corresponding to the product of the symbolic basis vectors using the formula

$$A_r \wedge b = \frac{1}{2}\left(A_r b + (-1)^r b A_r\right), \tag{2.33}$$

where A_r is a multivector of grade r and b is a vector. For our example basis the result is shown in Table 2.2.

```
1 = 1
a0 = a0
a1 = a1
a2 = a2
a0^a1 = {-(a0.a1)}1+a0a1
a0^a2 = {-(a0.a2)}1+a0a2
a1^a2 = {-(a1.a2)}1+a1a2
a0^a1^a2 = {-(a1.a2)}a0+{(a0.a2)}a1+{-(a0.a1)}a2+a0a1a2
```

Table 2.2: Bases blades in terms of bases.

The important thing to notice about Table 2.2 is that it is a triagonal (lower triangular) system of equations so that using a simple back substitution algorithm we can solve for the pseudo bases in terms of the blades giving Table 2.3.

```
1 = 1
a0 = a0
a1 = a1
a2 = a2
a0a1 = {(a0.a1)}1+a0^a1
a0a2 = {(a0.a2)}1+a0^a2
a1a2 = {(a1.a2)}1+a1^a2
a0a1a2 = {(a1.a2)}a0+{-(a0.a2)}a1+{(a0.a1)}a2+a0^a1^a2
```

Table 2.3: Bases in terms of basis blades.

Using Table 2.3 and simple substitution we can convert from a base multivector representation to a blade representation. Likewise, using Table 2.2 we can convert from blades to bases.

Using the blade representation it becomes simple to program functions that will calculate the grade projection, reverse, even, and odd multivector functions.

Note that in the multivector class `Mv` there is a class variable for each instantiation, `self.is_blade_rep`, that is set to `False` for a base representation and `True` for a blade representation. One needs to keep track of which representation is in use since various multivector operations require conversion from one representation to the other.

When the geometric product of two multivectors is calculated the module looks to see if either multivector is in blade representation. If either is the result of the geometric product is converted to a blade representation. One result of this is that if either of the multivectors is a simple vector (which is automatically a blade) the result will be in a blade representation. If `a` and `b` are vectors then the result `a*b` will be `(a.b)+a^b` or simply `a^b` if `(a.b) = 0`.

2.6 Outer and Inner Products, Left and Right Contractions

In geometric algebra any general multivector A can be decomposed into pure grade multivectors (a linear combination of blades of all the same order) so that in a n-dimensional vector space

$$A = \sum_{r=0}^{n} A_r \tag{2.34}$$

The geometric product of two pure grade multivectors A_r and B_s has the form

$$A_r B_s = \langle A_r B_s \rangle_{|r-s|} + \langle A_r B_s \rangle_{|r-s|+2} + \cdots + \langle A_r B_s \rangle_{r+s} \tag{2.35}$$

where $\langle\rangle_t$ projects the t grade components of the multivector argument. The inner and outer products of A_r and B_s are then defined to be

$$A_r \cdot B_s = \langle A_r B_s \rangle_{|r-s|} \tag{2.36}$$

$$A_r \wedge B_s = \langle A_r B_s \rangle_{r+s} \tag{2.37}$$

and

$$A \cdot B = \sum_{r,s>0} A_r \cdot B_s \tag{2.38}$$

$$A \wedge B = \sum_{r,s} A_r \wedge B_s \tag{2.39}$$

Likewise the right ($\lfloor$) and left ($\rfloor$) contractions are defined as

$$A_r \lfloor B_s = \left\{ \begin{array}{cc} \langle A_r B_s \rangle_{r-s} & r \geq s \\ 0 & r < s \end{array} \right\} \tag{2.40}$$

$$A_r \rfloor B_s = \left\{ \begin{array}{cc} \langle A_r B_s \rangle_{s-r} & s \geq r \\ 0 & s < r \end{array} \right\} \tag{2.41}$$

and

$$A \lfloor B = \sum_{r,s} A_r \lfloor B_s \tag{2.42}$$

$$A \rfloor B = \sum_{r,s} A_r \rfloor B_s \tag{2.43}$$

In the `Mv` class we have overloaded the `^` operator to represent the outer product so that instead of calling the outer product function we can write `mv1^mv2`. Due to the precedence rules for python it is *absolutely essential* to enclose outer products in parenthesis.

In the `Mv` class we have overloaded the `|` operator for the inner product, `>` operator for the right contraction, and `<` operator for the left contraction. Instead of calling the inner product function we can write `mv1|mv2`, `mv1>mv2`, or `mv1<mv2` respectively for the inner product, right contraction, or left contraction. Again, due to the precedence rules for python it is *absolutely essential*to enclose inner products and/or contractions in parenthesis.

2.7 Reverse of Multivector

If A is the geometric product of r vectors

$$A = a_1 \dots a_r \tag{2.44}$$

Then the reverse of A designated $A^\dagger$ is defined by

$$A^\dagger \equiv a_r \dots a_1. \tag{2.45}$$

The reverse is simply the product with the order of terms reversed. The reverse of a sum of products is defined as the sum of the reverses so that for a general multivector A we have

$$A^\dagger = \sum_{i=0}^{N} \langle A \rangle_i^\dagger \tag{2.46}$$

but

$$\langle A \rangle_i^\dagger = (-1)^{\frac{i(i-1)}{2}} \langle A \rangle_i \tag{2.47}$$

which is proved by expanding the blade bases in terms of orthogonal vectors and showing that eq. (2.47) holds for the geometric product of orthogonal vectors.

The reverse is important in the theory of rotations in n-dimensions. If R is the product of an even number of vectors and $RR^\dagger = 1$ then $RaR^\dagger$ is a composition of rotations of the vector a. If R is the product of two vectors then the plane that R defines is the plane of the rotation. That is to say that $RaR^\dagger$ rotates the component of a that is projected into the plane defined by a and b where $R = ab$. R may be written $R = e^{\frac{\theta}{2}U}$, where θ is the angle of rotation and U is a unit blade ($U^2 = \pm 1$) that defines the plane of rotation.

2.8 Reciprocal Frames

If we have M linearly independent vectors (a frame), $a_1, \ldots, a_M$, then the reciprocal frame is $a^1, \ldots, a^M$ where $a_i \cdot a^j = \delta_i^j$, δ_i^j is the Kronecker delta (zero if $i \neq j$ and one if $i = j$). The reciprocal frame is constructed as follows:

$$E_M = a_1 \wedge \ldots \wedge a_M \tag{2.48}$$

$$E_M^{-1} = \frac{E_M}{E_M^2} \tag{2.49}$$

Then

$$a^i = (-1)^{i-1} \left(a_1 \wedge \ldots \wedge \breve{a}_i \wedge \ldots \wedge a_M\right) E_M^{-1} \tag{2.50}$$

where $\breve{a}_i$ indicates that a_i is to be deleted from the product. In the standard notation if a vector is denoted with a subscript the reciprocal vector is denoted with a superscript. The set of reciprocal vectors will be calculated if a coordinate set is given when a geometric algebra is instantiated since they are required for geometric differentiation when the `Ga` member function `Ga.mvr()` is called to return the reciprocal basis in terms of the basis vectors.

2.9 Manifolds and Submanifolds

A m-dimensional vector manifold[1], $\mathcal{M}$, is defined by a coordinate tuple (tuples are indicated by the vector accent " $\vec{}$ ")

$$\vec{x} = \left(x^1, \ldots, x^m\right), \tag{2.51}$$

and the differentiable mapping (U^m is an m-dimensional subset of $\Re^m$)

$$\boldsymbol{e}^{\mathcal{M}}\left(\vec{x}\right) : U^m \subseteq \Re^m \to \mathcal{V}, \tag{2.52}$$

where $\mathcal{V}$ is a vector space with an inner product[2] $(\cdot)$ and is of $\dim\left(\mathcal{V}\right) \geq m$.

Then a set of basis vectors for the tangent space of $\mathcal{M}$ at $\vec{x}$, $\mathcal{T}_{\vec{x}}\left(\mathcal{M}\right)$, are

$$\boldsymbol{e}_i^{\mathcal{M}} = \frac{\partial \boldsymbol{e}^{\mathcal{M}}}{\partial x^i} \tag{2.53}$$

and

$$g_{ij}^{\mathcal{M}}\left(\vec{x}\right) = \boldsymbol{e}_i^{\mathcal{M}} \cdot \boldsymbol{e}_j^{\mathcal{M}}. \tag{2.54}$$

A n-dimensional ($n \leq m$) submanifold $\mathcal{N}$ of $\mathcal{M}$ is defined by a coordinate tuple

$$\vec{u} = \left(u^1, \ldots, u^n\right), \tag{2.55}$$

and a differentiable mapping

$$\vec{x}\left(\vec{u}\right) : U^n \subseteq \Re^n \to U^m \subseteq \Re^m, \tag{2.56}$$

which induces a mapping

$$\boldsymbol{e}^{\mathcal{M}}\left(\vec{x}\left(\vec{u}\right)\right) : U^n \subseteq \Re^n \to \mathcal{V}. \tag{2.57}$$

Then the basis vectors for the tangent space $\mathcal{T}_{\vec{u}}\left(\mathcal{N}\right)$ are (using $\boldsymbol{e}^{\mathcal{N}}\left(\vec{u}\right) = \boldsymbol{e}^{\mathcal{M}}\left(\vec{x}\left(\vec{u}\right)\right)$ and the chain rule)

$$\boldsymbol{e}_i^{\mathcal{N}}\left(\vec{u}\right) = \frac{\partial \boldsymbol{e}^{\mathcal{N}}\left(\vec{u}\right)}{\partial u^i} = \frac{\partial \boldsymbol{e}^{\mathcal{M}}\left(\vec{x}\right)}{\partial x^j}\frac{\partial x^j}{\partial u^i} = \boldsymbol{e}_j^{\mathcal{M}}\left(\vec{x}\left(\vec{u}\right)\right)\frac{\partial x^j}{\partial u^i}, \tag{2.58}$$

and

$$g_{ij}^{\mathcal{N}}\left(\vec{u}\right) = \frac{\partial x^k}{\partial u^i}\frac{\partial x^l}{\partial u^j} g_{kl}^{\mathcal{M}}\left(\vec{x}\left(\vec{u}\right)\right). \tag{2.59}$$

[1] By the manifold embedding theorem any m-dimensional manifold is isomorphic to a m-dimensional vector manifold

[2] This product in not necessarily positive definite.

Going back to the base manifold, $\mathcal{M}$, note that the mapping $\boldsymbol{e}^{\mathcal{M}}(\vec{x}) : U^n \subseteq \Re^n \to \mathcal{V}$ allows us to calculate an unnormalized pseudo-scalar for $\mathcal{T}_{\vec{x}}(\mathcal{M})$,

$$I^{\mathcal{M}}(\vec{x}) = \boldsymbol{e}_1^{\mathcal{M}}(\vec{x}) \wedge \ldots \wedge \boldsymbol{e}_m^{\mathcal{M}}(\vec{x}). \tag{2.60}$$

With the pseudo-scalar we can define a projection operator from $\mathcal{V}$ to the tangent space of $\mathcal{M}$ by

$$P_{\vec{x}}(\boldsymbol{v}) = (\boldsymbol{v} \cdot I^{\mathcal{M}}(\vec{x}))\left(I^{\mathcal{M}}(\vec{x})\right)^{-1} \ \forall\ \boldsymbol{v} \in \mathcal{V}. \tag{2.61}$$

In fact for each tangent space $\mathcal{T}_{\vec{x}}(\mathcal{M})$ we can define a geometric algebra $\mathcal{G}(\mathcal{T}_{\vec{x}}(\mathcal{M}))$ with pseudo-scalar $I^{\mathcal{M}}$ so that if $A \in \mathcal{G}(\mathcal{V})$ then

$$P_{\vec{x}}(A) = \left(A \cdot I^{\mathcal{M}}(\vec{x})\right)\left(I^{\mathcal{M}}(\vec{x})\right)^{-1} \in \mathcal{G}(\mathcal{T}_{\vec{x}}(\mathcal{M}))\ \forall\ A \in \mathcal{G}(\mathcal{V}) \tag{2.62}$$

and similarly for the submanifold $\mathcal{N}$.

If the embedding $\boldsymbol{e}^{\mathcal{M}}(\vec{x}) : U^n \subseteq \Re^n \to \mathcal{V}$ is not given, but the metric tensor $g_{ij}^{\mathcal{M}}(\vec{x})$ is given the geometric algebra of the tangent space can be constructed. Also the derivatives of the basis vectors of the tangent space can be calculated from the metric tensor using the Christoffel symbols, $\Gamma_{ij}^k(\vec{u})$, where the derivatives of the basis vectors are given by

$$\frac{\partial \boldsymbol{e}_j^{\mathcal{M}}}{\partial x^i} = \Gamma_{ij}^k(\vec{u})\, \boldsymbol{e}_k^{\mathcal{M}}. \tag{2.63}$$

If we have a submanifold, $\mathcal{N}$, defined by eq. (2.56) we can calculate the metric of $\mathcal{N}$ from eq. (2.59) and hence construct the geometric algebra and calculus of the tangent space, $\mathcal{T}_{\vec{u}}(\mathcal{N}) \subseteq \mathcal{T}_{\vec{x}(\vec{u})}(\mathcal{M})$.

If the base manifold is normalized (use the hat symbol to denote normalized tangent vectors, $\hat{\boldsymbol{e}}_i^{\mathcal{M}}$, and the resulting metric tensor, $\hat{g}_{ij}^{\mathcal{M}}$) we have $\hat{\boldsymbol{e}}_i^{\mathcal{M}} \cdot \hat{\boldsymbol{e}}_i^{\mathcal{M}} = \pm 1$ and $\hat{g}_{ij}^{\mathcal{M}}$ does not posses enough information to calculate $g_{ij}^{\mathcal{N}}$. In that case we need to know $g_{ij}^{\mathcal{M}}$, the metric tensor of the base manifold before normalization. Likewise, for the case of a vector manifold unless the mapping, $\boldsymbol{e}^{\mathcal{M}}(\vec{x}) : U^m \subseteq \Re^m \to \mathcal{V}$, is constant the tangent vectors and metric tensor can only be normalized after the fact (one cannot have a mapping that automatically normalizes all the tangent vectors).

2.10 Geometric Derivative

The directional derivative of a multivector field $F(x)$ is defined by (a is a vector and h is a scalar)

$$(a \cdot \nabla_x) F \equiv \lim_{h \to 0} \frac{F(x + ah) - F(x)}{h}. \tag{2.64}$$

Note that $a \cdot \nabla_x$ is a scalar operator. It will give a result containing only those grades that are already in F. $(a \cdot \nabla_x) F$ is the best linear approximation of $F(x)$ in the direction a. Equation (2.64) also defines the operator ∇_x which for the basis vectors, $\{\boldsymbol{e}_i\}$, has the representation (note that the $\{\boldsymbol{e}^j\}$ are reciprocal basis vectors)

$$\nabla_x F = \boldsymbol{e}^j \frac{\partial F}{\partial x^j} \tag{2.65}$$

If F_r is a r-grade multivector (if the independent vector, x, is obvious we suppress it in the notation and just write ∇) and $F_r = F_r^{i_1 \ldots i_r} \boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r}$ then

$$\nabla F_r = \frac{\partial F_r^{i_1 \ldots i_r}}{\partial x^j} \boldsymbol{e}^j \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right) \tag{2.66}$$

Note that $\boldsymbol{e}^j \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right)$ can only contain grades $r-1$ and $r+1$ so that ∇F_r also can only contain those grades. For a grade-r multivector F_r the inner (div) and outer (curl) derivatives are

$$\nabla \cdot F_r = \langle \nabla F_r \rangle_{r-1} = \boldsymbol{e}^j \cdot \frac{\partial F_r}{\partial x^j} \tag{2.67}$$

and

$$\nabla \wedge F_r = \langle \nabla F_r \rangle_{r+1} = \boldsymbol{e}^j \wedge \frac{\partial F_r}{\partial x^j} \tag{2.68}$$

For a general multivector function F the inner and outer derivatives are just the sum of the inner and outer derivatives of each grade of the multivector function.

2.10.1 Geometric Derivative on a Manifold

In the case of a manifold the derivatives of the $\boldsymbol{e}_i$'s are functions of the coordinates, $\{x^i\}$, so that the geometric derivative of a r-grade multivector field is (Einstein summation convention)

$$\begin{aligned} \nabla F_r &= \boldsymbol{e}^i \frac{\partial F_r}{\partial x^i} = \boldsymbol{e}^i \frac{\partial}{\partial x^i} \left(F_r^{i_1 \ldots i_r} \boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right) \\ &= \frac{\partial F_r^{i_1 \ldots i_r}}{\partial x^i} \boldsymbol{e}^i \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right) + F_r^{i_1 \ldots i_r} \boldsymbol{e}^i \frac{\partial}{\partial x^i} \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right) \end{aligned} \tag{2.69}$$

where the multivector functions $\boldsymbol{e}^i \dfrac{\partial}{\partial x^i} \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right)$ are the connection for the manifold.[3]

The directional (material/convective) derivative, $(v \cdot \nabla) F_r$ is given by

$$\begin{aligned}(v \cdot \nabla) F_r &= v^i \frac{\partial F_r}{\partial x^i} = v^i \frac{\partial}{\partial x^i} \left(F_r^{i_1 \ldots i_r} \boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right) \\ &= v^i \frac{\partial F_r^{i_1 \ldots i_r}}{\partial x^i} \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right) + v^i F_r^{i_1 \ldots i_r} \frac{\partial}{\partial x^i} \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right), \end{aligned} \tag{2.70}$$

so that the multivector connection functions for the directional derivative are $\dfrac{\partial}{\partial x^i} \left(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r} \right)$. Be careful and note that $(v \cdot \nabla) F_r \neq v \cdot (\nabla F_r)$ since the dot and geometric products are not associative with respect to one another ($v \cdot \nabla$ is a scalar operator).

2.10.2 Normalizing Basis for Derivatives

The basis vector set, $\{\boldsymbol{e}_i\}$, is not in general normalized. We define a normalized set of basis vectors, $\{\hat{\boldsymbol{e}}_i\}$, by

$$\hat{\boldsymbol{e}}_i = \frac{\boldsymbol{e}_i}{\sqrt{|\boldsymbol{e}_i^2|}} = \frac{\boldsymbol{e}_i}{|\boldsymbol{e}_i|}. \tag{2.71}$$

This works for all $\boldsymbol{e}_i^2 \neq 0$. Note that $\hat{\boldsymbol{e}}_i^2 = \pm 1$.

[3] We use the Christoffel symbols of the first kind to calculate the derivatives of the basis vectors and the product rule to calculate the derivatives of the basis blades where (http://en.wikipedia.org/wiki/Christoffel_symbols)

$$\Gamma_{ijk} = \frac{1}{2} \left(\frac{\partial g_{jk}}{\partial x^i} + \frac{\partial g_{ik}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^k} \right),$$

and

$$\frac{\partial \boldsymbol{e}_j}{\partial x^i} = \Gamma_{ijk} \boldsymbol{e}^k.$$

The Christoffel symbols of the second kind,

$$\Gamma_{ij}^k = \frac{1}{2} g^{kl} \left(\frac{\partial g_{li}}{\partial x^j} + \frac{\partial g_{lj}}{\partial x^i} - \frac{\partial g_{ij}}{\partial x^l} \right),$$

could also be used to calculate the derivatives in term of the original basis vectors, but since we need to calculate the reciprocal basis vectors for the geometric derivative it is more efficient to use the symbols of the first kind.

Thus the geometric derivative for a set of normalized basis vectors is (where $F_r = F_r^{i_1 \ldots i_r} \hat{\boldsymbol{e}}_{i_1} \wedge \ldots \wedge \hat{\boldsymbol{e}}_{i_r}$ and [no summation] $\hat{F}_r^{i_1 \ldots i_r} = F_r^{i_1 \ldots i_r} |\hat{\boldsymbol{e}}_{i_1}| \ldots |\hat{\boldsymbol{e}}_{i_r}|$).

$$\nabla F_r = \boldsymbol{e}^i \frac{\partial F_r}{\partial x^i} = \frac{\partial F_r^{i_1 \ldots i_r}}{\partial x^i} \boldsymbol{e}^i \left(\hat{\boldsymbol{e}}_{i_1} \wedge \ldots \wedge \hat{\boldsymbol{e}}_{i_r} \right) + F_r^{i_1 \ldots i_r} \boldsymbol{e}^i \frac{\partial}{\partial x^i} \left(\hat{\boldsymbol{e}}_{i_1} \wedge \ldots \wedge \hat{\boldsymbol{e}}_{i_r} \right). \tag{2.72}$$

To calculate $\boldsymbol{e}^i$ in terms of the $\hat{\boldsymbol{e}}_i$'s we have

$$\begin{aligned} \boldsymbol{e}^i &= g^{ij} \boldsymbol{e}_j \\ \boldsymbol{e}^i &= g^{ij} |\boldsymbol{e}_j| \, \hat{\boldsymbol{e}}_j. \end{aligned} \tag{2.73}$$

This is the general (non-orthogonal) formula. If the basis vectors are orthogonal then (no summation over repeated indexes)

$$\begin{aligned} \boldsymbol{e}^i &= g^{ii} |\boldsymbol{e}_i| \, \hat{\boldsymbol{e}}_i \\ \boldsymbol{e}^i &= \frac{|\boldsymbol{e}_i|}{g_{ii}} \hat{\boldsymbol{e}}_i = \frac{|\hat{\boldsymbol{e}}_i|}{\boldsymbol{e}_i^2} \hat{\boldsymbol{e}}_i. \end{aligned} \tag{2.74}$$

Additionally, one can calculate the connection of the normalized basis as follows

$$\begin{aligned} \frac{\partial (|\boldsymbol{e}_i| \, \hat{\boldsymbol{e}}_i)}{\partial x^j} &= \frac{\partial \boldsymbol{e}_i}{\partial x^j}, \\ \frac{\partial |\boldsymbol{e}_i|}{\partial x^j} \hat{\boldsymbol{e}}_i + |\boldsymbol{e}_i| \frac{\partial \hat{\boldsymbol{e}}_i}{\partial x^j} &= \frac{\partial \boldsymbol{e}_i}{\partial x^j}, \\ \frac{\partial \hat{\boldsymbol{e}}_i}{\partial x^j} &= \frac{1}{|\boldsymbol{e}_i|} \left(\frac{\partial \boldsymbol{e}_i}{\partial x^j} - \frac{\partial |\boldsymbol{e}_i|}{\partial x^j} \hat{\boldsymbol{e}}_i \right), \\ &= \frac{1}{|\boldsymbol{e}_i|} \frac{\partial \boldsymbol{e}_i}{\partial x^j} - \frac{1}{|\boldsymbol{e}_i|} \frac{\partial |\boldsymbol{e}_i|}{\partial x^j} \hat{\boldsymbol{e}}_i, \\ &= \frac{1}{|\boldsymbol{e}_i|} \frac{\partial \boldsymbol{e}_i}{\partial x^j} - \frac{1}{2 g_{ii}} \frac{\partial g_{ii}}{\partial x^j} \hat{\boldsymbol{e}}_i, \end{aligned} \tag{2.75}$$

where $\dfrac{\partial \boldsymbol{e}_i}{\partial x^j}$ is expanded in terms of the $\hat{\boldsymbol{e}}_i$'s.

2.10.3 Linear Differential Operators

First a note on partial derivative notation. We shall use the following notation for a partial derivative where the manifold coordinates are $x_1, \ldots, x_n$:

$$\frac{\partial^{j_1 + \cdots + j_n}}{\partial x_1^{j_1} \ldots \partial x_n^{j_n}} = \partial_{j_1 \ldots j_n}. \tag{2.76}$$

If $j_k = 0$ the partial derivative with respect to the k^{th} coordinate is not taken. If the $j_k = 0$ for all $1 \leq k \leq n$ then the partial derivative operator is the scalar one. If we consider a partial derivative where the x's are not in normal order such as

$$\frac{\partial^{j_1+\cdots+j_n}}{\partial x_{i_1}^{j_1} \ldots \partial x_{i_n}^{j_n}},$$

and the i_k's are not in ascending order. The derivative can always be put in the form in eq (2.76) since the order of differentiation does not change the value of the partial derivative (for the smooth functions we are considering). Additionally, using our notation the product of two partial derivative operations is given by

$$\partial_{i_1 \ldots i_n} \partial_{j_1 \ldots j_n} = \partial_{i_1+j_1,\ldots,i_n+j_n}. \tag{2.77}$$

A general general multivector linear differential operator is a linear combination of multivectors and partial derivative operators denoted by (in all of this section we will use the Einstein summation convention)

$$D \equiv D^{i_1 \ldots i_n} \partial_{i_1 \ldots i_n}. \tag{2.78}$$

Equation (2.78) is the normal form of the differential operator in that the partial derivative operators are written to the right of the multivector coefficients and do not operate upon the multivector coefficients. The operator of eq (2.78) can operate on mulitvector functions, returning a multivector function via the following definitions.

F as (Einstein summation convention)

$$D \circ F = D^{j_1 \ldots j_n} \circ \partial_{j_1 \ldots j_n} F, \tag{2.79}$$

or

$$F \circ D = \partial_{j_1 \ldots j_n} F \circ D^{j_1 \ldots j_n}, \tag{2.80}$$

where the $D^{j_1 \ldots j_n}$ are multivector functions and $\circ$ is any of the multivector multiplicative operations.

Equations (2.79) and (2.80) are not the most general multivector linear differential operators, the most general would be

$$D\left(F\right) = D^{j_1 \ldots j_n}\left(\partial_{j_1 \ldots j_n} F\right), \tag{2.81}$$

where $D^{j_1 \ldots j_n}\left(\right)$ are linear multivector functionals.

The definition of the sum of two differential operators is obvious since any multivector operator, $\circ$, is a bilinear operator ($(D_A + D_B) \circ F = D_A \circ F + D_B \circ F$), the product of two differential

operators D_A and D_B operating on a multivector function F is defined to be ($\circ_1$ and $\circ_2$ are any two multivector multiplicative operations)

$$\begin{aligned}(D_A \circ_1 D_B) \circ_2 F &\equiv \left(D_A^{i_1\ldots i_n} \circ_1 \partial_{i_1\ldots i_n}\left(D_B^{j_1\ldots j_n}\partial_{j_1\ldots j_n}\right)\right) \circ_2 F \\ &= \left(D_A^{i_1\ldots i_n} \circ_1 \left(\left(\partial_{i_1\ldots i_n} D_B^{j_1\ldots j_n}\right)\partial_{j_1\ldots j_n} + D_B^{j_1\ldots j_n}\right)\partial_{i_1+j_1,\ldots,i_n+j_n}\right) \circ_2 F \\ &= \left(D_A^{i_1\ldots i_n} \circ_1 \left(\partial_{i_1\ldots i_n} D_B^{j_1\ldots j_n}\right)\right) \circ_2 \partial_{j_1\ldots j_n} F + \left(D_A^{i_1\ldots i_n} \circ_1 D_B^{j_1\ldots j_n}\right) \circ_2 \partial_{i_1+j_1,\ldots,i_n+j_n} F,\end{aligned}$$

where we have used the fact that the ∂ operator is a scalar operator and commutes with $\circ_1$ and $\circ_2$.

Thus for a pure operator product $D_A \circ D_B$ we have

$$D_A \circ D_B = \left(D_A^{i_1\ldots i_n} \circ \left(\partial_{i_1\ldots i_n} D_B^{j_1\ldots j_n}\right)\right)\partial_{j_1\ldots j_n} + \left(D_A^{i_1\ldots i_n} \circ_1 D_B^{j_1\ldots j_n}\right)\partial_{i_1+j_1,\ldots,i_n+j_n} \tag{2.82}$$

and the form of eq (2.82) is the same as eq(2.79). The basis of eq (2.82) is that the ∂ operator operates on all object to the right of it as products so that the product rule must be used in all differentiations. Since eq (2.82) puts the product of two differential operators in standard form we also evaluate $F \circ_2 (D_A \circ_1 D_B)$.

We now must distinguish between the following cases. If D is a differential operator and F a multivector function should $D \circ F$ and $F \circ D$ return a differential operator or a multivector. In order to be consistent with the standard vector analysis we have $D \circ F$ return a multivector and $F \circ D$ return a differential operator. The we define the complementary differential operator $\bar{D}$ which is identical to D except that $\bar{D} \circ F$ returns a differential operator according to eq (2.82)[4] and $F \circ \bar{D}$ returns a multivector according to eq (2.80).

A general differential operator is built from repeated applications of the basic operator building blocks $\left(\bar{\nabla} \circ A\right)$, $\left(A \circ \bar{\nabla}\right)$, $\left(\bar{\nabla} \circ \bar{\nabla}\right)$, and $\left(A \pm \bar{\nabla}\right)$. Both ∇ and $\bar{\nabla}$ are represented by the operator

$$\nabla = \bar{\nabla} = e^i \frac{\partial}{\partial x^i}, \tag{2.83}$$

but are flagged to produce the appropriate result.

In the our notation the directional derivative operator is $a \cdot \nabla$, the Laplacian $\nabla \cdot \nabla$ and the expression for the Riemann tensor, R^i_{jkl}, is

$$(\nabla \wedge \nabla)\, \boldsymbol{e}^i = \frac{1}{2} R^i_{jkl} \left(\boldsymbol{e}^j \wedge \boldsymbol{e}^k\right) \boldsymbol{e}^l. \tag{2.84}$$

[4] In this case $D_B^{j_1\ldots j_n} = F$ and $\partial_{j_1\ldots j_n} = 1$.

We would use the complement if we wish a quantum mechanical type commutator defining

$$[x, \nabla] \equiv x\nabla - \bar{\nabla}x, \tag{2.85}$$

or if we wish to simulate the dot notation (Doran and Lasenby)

$$\dot{F}\dot{\nabla} = F\bar{\nabla}. \tag{2.86}$$

2.11 Linear Transformations/Outermorphisms

In the tangent space of a manifold, $\mathcal{M}$, (which is a vector space) a linear transformation is the mapping $\underline{T}$: $\mathcal{T}_{\vec{x}}(\mathcal{M}) \to \mathcal{T}_{\vec{x}}(\mathcal{M})$ (we use an underline to indicate a linear transformation) where for all $x, y \in \mathcal{T}_{\vec{x}}(\mathcal{M})$ and $\alpha \in \Re$ we have

$$\underline{T}(x+y) = \underline{T}(x) + \underline{T}(y) \tag{2.87}$$
$$\underline{T}(\alpha x) = \alpha \underline{T}(x) \tag{2.88}$$

The outermorphism induced by $\underline{T}$ is defined for $x_1, \ldots, x_r \in \mathcal{T}_{\vec{x}}(\mathcal{M})$ where $r \leq \dim(\mathcal{T}_{\vec{x}}(\mathcal{M}))$

$$\underline{T}(x_1 \wedge \ldots \wedge x_r) \equiv \underline{T}(x_1) \wedge \ldots \wedge \underline{T}(x_r) \tag{2.89}$$

If I is the pseudo scalar for $\mathcal{T}_{\vec{x}}(\mathcal{M})$ we also have the following definitions for determinate, trace, and adjoint ($\overline{T}$) of $\underline{T}$

$$\underline{T}(I) \equiv \det(\underline{T})\, I,^{5} \tag{2.90}$$
$$\mathrm{tr}(\underline{T}) \equiv \nabla_y \cdot \underline{T}(y),^{6} \tag{2.91}$$
$$x \cdot \overline{T}(y) \equiv y \cdot \underline{T}(x). \tag{2.92}$$

If $\{\boldsymbol{e}_i\}$ is a basis for $\mathcal{T}_{\vec{x}}(\mathcal{M})$ then we can represent $\underline{T}$ with the matrix $\underline{T}_i^j$ used as follows (Einstein summation convention as usual) -

$$\underline{T}(\boldsymbol{e}_i) = \underline{T}_i^j \boldsymbol{e}_j, \tag{2.93}$$

where

$$\underline{T}_i^j = \boldsymbol{e}^j \cdot \underline{T}(\boldsymbol{e}_i). \tag{2.94}$$

[5] Since $\underline{T}$ is linear we do not require $I^2 = \pm 1$.

[6] In this case y is a vector in the tangent space and not a coordinate vector so that the basis vectors are *not* a function of y.

The let $\left(\underline{T}^{-1}\right)_m^n$ be the inverse matrix of $\underline{T}_i^j$ so that $\left(\underline{T}^{-1}\right)_m^k \underline{T}_k^j = \delta_m^j$ and

$$\underline{T}^{-1}\left(a^i \boldsymbol{e}_i\right) = a^i \left(\underline{T}^{-1}\right)_i^j \boldsymbol{e}_j \tag{2.95}$$

and calculate

$$\begin{aligned}
\underline{T}^{-1}\left(\underline{T}\left(a\right)\right) &= \underline{T}^{-1}\left(\underline{T}\left(a^i \boldsymbol{e}_i\right)\right) \\
&= \underline{T}^{-1}\left(a^i \underline{T}_i^j \boldsymbol{e}_j\right) \\
&= a^i \left(\underline{T}^{-1}\right)_i^j \underline{T}_j^k \boldsymbol{e}_k \\
&= a^i \delta_i^j \boldsymbol{e}_j = a^i \boldsymbol{e}_i = a.
\end{aligned} \tag{2.96}$$

Thus if eq 2.94 is used to define the $\underline{T}_i^j$ then the linear transformation defined by the matrix $\left(\underline{T}^{-1}\right)_m^n$ is the inverse of $\underline{T}$.

In eq. (2.93) the matrix, $\underline{T}_i^j$, only has it's usual meaning if the $\{\boldsymbol{e}_i\}$ form an orthonormal Euclidean basis (Minkowski spaces not allowed). Equations (2.90) through (2.92) become

$$\det\left(\underline{T}\right) = \underline{T}\left(\boldsymbol{e}_1 \wedge \ldots \wedge \boldsymbol{e}_n\right)\left(\boldsymbol{e}_1 \wedge \ldots \wedge \boldsymbol{e}_n\right)^{-1}, \tag{2.97}$$

$$\mathrm{tr}\left(\underline{T}\right) = \underline{T}_i^i, \tag{2.98}$$

$$\overline{T}_j^i = g^{il} g_{jp} \underline{T}_l^p. \tag{2.99}$$

A important form of linear transformation with a simple representation is the spinor transformation. If S is an even multivector we have $SS^\dagger = \rho^2$, where ρ^2 is a scalar. Then the spinor transformation is given by (v is a vector)

$$S\left(v\right) = SvS^\dagger \tag{2.100}$$

and

$$S^{-1}\left(v\right) = \frac{S^\dagger v S}{\rho^4}. \tag{2.101}$$

Thus

$$\begin{aligned}
S^{-1}\left(S\left(v\right)\right) &= \frac{S^\dagger S v S^\dagger S}{\rho^4} \\
&= \frac{\rho^2 v \rho^2}{\rho^4} \\
&= v.
\end{aligned} \tag{2.102}$$

2.12 Multilinear Functions (Tensors)

2.12.1 Multilinear Functions

A multivector multilinear function[7] is a multivector function $T(A_1, \ldots, A_r)$ that is linear in each of it arguments[8] (it could be implicitly non-linearly dependent on a set of additional arguments such as the position coordinates, but we only consider the linear arguments). T is a *tensor* of degree r if each variable A_j is restricted to the vector space $\mathcal{V}_n$. More generally if each $A_j \in \mathcal{G}(\mathcal{V}_n)$ (the geometric algebra of $\mathcal{V}_n$), we call T an *extensor* of degree-r on $\mathcal{G}(\mathcal{V}_n)$.

If the values of $T(a_1, \ldots, a_r)$ $(a_j \in \mathcal{V}_n \; \forall \; 1 \leq j \leq r)$ are s-vectors (pure grade s multivectors in $\mathcal{G}(\mathcal{V}_n)$) we say that T has grade s and rank $r + s$. A tensor of grade zero is called a *multilinear form*.

In the normal definition of tensors as multilinear functions the tensor is defined as a mapping

$$T : \bigtimes_{i=1}^{r} \mathcal{V}_i \to \Re,$$

so that the standard tensor definition is an example of a grade zero degree/rank r tensor in our definition.

2.12.2 Algebraic Operations

The properties of tensors are ($\alpha \in \Re$, $a_j, b \in \mathcal{V}_n$, T and S are tensors of rank r, and $\circ$ is any multivector multiplicative operation)

$$T(a_1, \ldots, \alpha a_j, \ldots, a_r) = \alpha T(a_1, \ldots, a_j, \ldots, a_r), \tag{2.103}$$

$$T(a_1, \ldots, a_j + b, \ldots, a_r) = T(a_1, \ldots, a_j, \ldots, a_r) + T(a_1, \ldots, a_{j-1}, b, a_{j+1}, \ldots, a_r), \tag{2.104}$$

$$(T \pm S)(a_1, \ldots, a_r) \equiv T(a_1, \ldots, a_r) \pm S(a_1, \ldots, a_r). \tag{2.105}$$

Now let T be of rank r and S of rank s then the product of the two tensors is

$$(T \circ S)(a_1, \ldots, a_{r+s}) \equiv T(a_1, \ldots, a_r) \circ S(a_{r+1}, \ldots, a_{r+s}), \tag{2.106}$$

where "$\circ$" is any multivector multiplicative operation.

[7] We are following the treatment of Tensors in section 3–10 of [2].

[8] We assume that the arguments are elements of a vector space or more generally a geometric algebra so that the concept of linearity is meaningful.

2.12.3 Covariant, Contravariant, and Mixed Representations

The arguments (vectors) of the multilinear function can be represented in terms of the basis vectors or the reciprocal basis vectors

$$a_j = a^{i_j} \boldsymbol{e}_{i_j}, \tag{2.107}$$

$$= a_{i_j} \boldsymbol{e}^{i_j}. \tag{2.108}$$

Equation (2.107) gives a_j in terms of the basis vectors and eq (2.108) in terms of the reciprocal basis vectors. The index j refers to the argument slot and the indices i_j the components of the vector in terms of the basis. The Einstein summation convention is used throughout. The covariant representation of the tensor is defined by

$$T_{i_1 \ldots i_r} \equiv T\left(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}_{i_r}\right) \tag{2.109}$$

$$\begin{aligned} T\left(a_1, \ldots, a_r\right) &= T\left(a^{i_1} \boldsymbol{e}_{i_1}, \ldots, a^{i_r} \boldsymbol{e}_{i_r}\right) \\ &= T\left(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}_{i_r}\right) a^{i_1} \ldots a^{i_r} \\ &= T_{i_1 \ldots i_r} a^{i_1} \ldots a^{i_r}. \end{aligned} \tag{2.110}$$

Likewise for the contravariant representation

$$T^{i_1 \ldots i_r} \equiv T\left(\boldsymbol{e}^{i_1}, \ldots, \boldsymbol{e}^{i_r}\right) \tag{2.111}$$

$$\begin{aligned} T\left(a_1, \ldots, a_r\right) &= T\left(a_{i_1} \boldsymbol{e}^{i_1}, \ldots, a_{i_r} \boldsymbol{e}^{i_r}\right) \\ &= T\left(\boldsymbol{e}^{i_1}, \ldots, \boldsymbol{e}^{i_r}\right) a_{i_1} \ldots a_{i_r} \\ &= T^{i_1 \ldots i_r} a_{i_1} \ldots a_{i_r}. \end{aligned} \tag{2.112}$$

One could also have a mixed representation

$$T_{i_1 \ldots i_s}{}^{i_{s+1} \ldots i_r} \equiv T\left(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}_{i_s}, \boldsymbol{e}^{i_{s+1}} \ldots \boldsymbol{e}^{i_r}\right) \tag{2.113}$$

$$\begin{aligned} T\left(a_1, \ldots, a_r\right) &= T\left(a^{i_1} \boldsymbol{e}_{i_1}, \ldots, a^{i_s} \boldsymbol{e}_{i_s}, a_{i_{s+1}} \boldsymbol{e}^{i_s} \ldots, a_{i_r} \boldsymbol{e}^{i_r}\right) \\ &= T\left(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}_{i_s}, \boldsymbol{e}^{i_{s+1}}, \ldots, \boldsymbol{e}^{i_r}\right) a^{i_1} \ldots a^{i_s} a_{i_{s+1}}, \ldots a^{i_r} \\ &= T_{i_1 \ldots i_s}{}^{i_{s+1} \ldots i_r} a^{i_1} \ldots a^{i_s} a_{i_{s+1}} \ldots a^{i_r}. \end{aligned} \tag{2.114}$$

In the representation of T one could have any combination of covariant (lower) and contravariant (upper) indexes.

To convert a covariant index to a contravariant index simply consider

$$T\left(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}^{i_j}, \ldots, \boldsymbol{e}_{i_r}\right) = T\left(\boldsymbol{e}_{i_1}, \ldots, g^{i_j k_j} \boldsymbol{e}_{k_j}, \ldots, \boldsymbol{e}_{i_r}\right)$$

$$
\begin{aligned}
&= g^{i_j k_j} T\left(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}_{k_j}, \ldots, \boldsymbol{e}_{i_r}\right) \\
T_{i_1 \ldots}{}^{i_j}{}_{\ldots i_r} &= g^{i_j k_j} T_{i_1 \ldots i_j \ldots i_r} .
\end{aligned} \tag{2.115}
$$

Similarly one could lower an upper index with $g_{i_j k_j}$.

2.12.4 Contraction and Differentiation

The contraction of a tensor between the j^{th} and k^{th} variables (slots) is

$$
T\left(a_i, \ldots, a_{j-1}, \nabla_{a_k}, a_{j+1}, \ldots, a_r\right) = \nabla_{a_j} \cdot \left(\nabla_{a_k} T\left(a_1, \ldots, a_r\right)\right). \tag{2.116}
$$

This operation reduces the rank of the tensor by two. This definition gives the standard results for *metric contraction* which is proved as follows for a rank r grade zero tensor (the circumflex "$\breve{}$" indicates that a term is to be deleted from the product).

$$
T\left(a_1, \ldots, a_r\right) = a^{i_1} \ldots a^{i_r} T_{i_1 \ldots i_r} \tag{2.117}
$$

$$
\begin{aligned}
\nabla_{a_j} T &= \boldsymbol{e}^{l_j} a^{i_1} \ldots \left(\partial_{a^{l_j}} a^{i_j}\right) \ldots a_{i_r} T_{i_1 \ldots i_r} \\
&= \boldsymbol{e}^{l_j} \delta^{i_j}_{l_j} a^{i_1} \ldots \breve{a}^{i_j} \ldots a^{i_r} T_{i_1 \ldots i_r}
\end{aligned} \tag{2.118}
$$

$$
\begin{aligned}
\nabla_{a_m} \cdot \left(\nabla_{a_j} T\right) &= \boldsymbol{e}^{k_m} \cdot \boldsymbol{e}^{l_j} \delta^{i_j}_{l_j} a^{i_1} \ldots \breve{a}^{i_j} \ldots \left(\partial_{a^{k_m}} a^{i_m}\right) \ldots a^{i_r} T_{i_1 \ldots i_r} \\
&= g^{k_m l_j} \delta^{i_j}_{l_j} \delta^{i_m}_{k_m} a^{i_1} \ldots \breve{a}^{i_j} \ldots \breve{a}^{i_m} \ldots a^{i_r} T_{i_1 \ldots i_r} \\
&= g^{i_m i_j} a^{i_1} \ldots \breve{a}^{i_j} \ldots \breve{a}^{i_m} \ldots a^{i_r} T_{i_1 \ldots i_j \ldots i_m \ldots i_r} \\
&= g^{i_j i_m} a^{i_1} \ldots \breve{a}^{i_j} \ldots \breve{a}^{i_m} \ldots a^{i_r} T_{i_1 \ldots i_j \ldots i_m \ldots i_r} \\
&= \left(g^{i_j i_m} T_{i_1 \ldots i_j \ldots i_m \ldots i_r}\right) a^{i_1} \ldots \breve{a}^{i_j} \ldots \breve{a}^{i_m} \ldots a^{i_r}
\end{aligned} \tag{2.119}
$$

Equation (2.119) is the correct formula for the metric contraction of a tensor.

If we have a mixed representation of a tensor, $T_{i_1 \ldots}{}^{i_j}{}_{\ldots i_k \ldots i_r}$, and wish to contract between an upper and lower index (i_j and i_k) first lower the upper index and then use eq (2.119) to contract the result. Remember lowering the index does *not* change the tensor, only the *representation* of the tensor, while contraction results in a *new* tensor. First lower index

$$
T_{i_1 \ldots}{}^{i_j}{}_{\ldots i_k \ldots i_r} \xrightarrow{\text{Lower Index}} g_{i_j k_j} T_{i_1 \ldots}{}^{k_j}{}_{\ldots i_k \ldots i_r} \tag{2.120}
$$

Now contract between i_j and i_k and use the properties of the metric tensor.

$$
g_{i_j k_j} T_{i_1 \ldots}{}^{k_j}{}_{\ldots i_k \ldots i_r} \xrightarrow{\text{Contract}} g^{i_j i_k} g_{i_j k_j} T_{i_1 \ldots}{}^{k_j}{}_{\ldots i_k \ldots i_r}
$$

$$= \delta^{i_k}_{k_j} T_{i_1 \ldots \quad \ldots i_k \ldots i_r}^{\quad k_j} . \tag{2.121}$$

Equation (2.121) is the standard formula for contraction between upper and lower indexes of a mixed tensor.

Finally if $T(a_1, \ldots, a_r)$ is a tensor field (implicitly a function of position) the tensor derivative is defined as

$$T(a_1, \ldots, a_r; a_{r+1}) \equiv (a_{r+1} \cdot \nabla) T(a_1, \ldots, a_r), \tag{2.122}$$

assuming the a^{i_j} coefficients are not a function of the coordinates.

This gives for a grade zero rank r tensor

$$\begin{aligned} (a_{r+1} \cdot \nabla) T(a_1, \ldots, a_r) &= a^{i_{r+1}} \partial_{x^{i_{r+1}}} a^{i_1} \ldots a^{i_r} T_{i_1 \ldots i_r}, \\ &= a^{i_1} \ldots a^{i_r} a^{i_{r+1}} \partial_{x^{i_{r+1}}} T_{i_1 \ldots i_r}. \end{aligned} \tag{2.123}$$

2.12.5 From Vector to Tensor

A rank one tensor is a vector since it satisfies all the axioms for a vector space, but a vector in not necessarily a tensor since not all vectors are multilinear (actually in the case of vectors a linear function) functions. However, there is a simple isomorphism between vectors and rank one tensors defined by the mapping $v(a) : \mathcal{V} \to \Re$ such that if $v, a \in \mathcal{V}$

$$v(a) \equiv v \cdot a. \tag{2.124}$$

So that if $v = v^i \boldsymbol{e}_i = v_i \boldsymbol{e}^i$ the covariant and contravariant representations of v are (using $\boldsymbol{e}^i \cdot \boldsymbol{e}_j = \delta^i_j$)

$$v(a) = v_i a^i = v^i a_i. \tag{2.125}$$

2.12.6 Parallel Transport and Covariant Derivatives

The covariant derivative of a tensor field $T(a_1, \ldots, a_r; x)$ (x is the coordinate vector of which T can be a non-linear function) in the direction a_{r+1} is (remember $a_j = a^k_j \boldsymbol{e}_k$ and the $\boldsymbol{e}_k$ can be functions of x) the directional derivative of $T(a_1, \ldots, a_r; x)$ where all the arguments of T are parallel transported. The definition of parallel transport is if a and b are tangent vectors in the tangent spaced of the manifold then

$$(a \cdot \nabla_x) b = 0 \tag{2.126}$$

if b is parallel transported. Since $b = b^i \boldsymbol{e}_i$ and the derivatives of $\boldsymbol{e}_i$ are functions of the x^i's then the b^i's are also functions of the x^i's so that in order for eq (2.126) to be satisfied we have

$$\begin{aligned}
(a \cdot \nabla_x)\, b =& a^i \partial_{x^i} \left(b^j \boldsymbol{e}_j \right) \\
=& a^i \left(\left(\partial_{x^i} b^j \right) \boldsymbol{e}_j + b^j \partial_{x^i} \boldsymbol{e}_j \right) \\
=& a^i \left(\left(\partial_{x^i} b^j \right) \boldsymbol{e}_j + b^j \Gamma_{ij}^k \boldsymbol{e}_k \right) \\
=& a^i \left(\left(\partial_{x^i} b^j \right) \boldsymbol{e}_j + b^k \Gamma_{ik}^j \boldsymbol{e}_j \right) \\
=& a^i \left(\left(\partial_{x^i} b^j \right) + b^k \Gamma_{ik}^j \right) \boldsymbol{e}_j = 0.
\end{aligned} \tag{2.127}$$

Thus for b to be parallel transported we must have

$$\partial_{x^i} b^j = -b^k \Gamma_{ik}^j. \tag{2.128}$$

The geometric meaning of parallel transport is that for an infinitesimal rotation and dilation of the basis vectors (cause by infinitesimal changes in the x^i's) the direction and magnitude of the vector b does not change.

If we apply eq (2.128) along a parametric curve defined by $x^j(s)$ we have

$$\begin{aligned}
\frac{db^j}{ds} =& \frac{dx^i}{ds} \frac{\partial b^j}{\partial x^i} \\
=& -b^k \frac{dx^i}{ds} \Gamma_{ik}^j,
\end{aligned} \tag{2.129}$$

and if we define the initial conditions $b^j(0)\, \boldsymbol{e}_j$. Then eq (2.129) is a system of first order linear differential equations with initial conditions and the solution, $b^j(s)\, \boldsymbol{e}_j$, is the parallel transport of the vector $b^j(0)\, \boldsymbol{e}_j$.

An equivalent formulation for the parallel transport equation is to let $\gamma(s)$ be a parametric curve in the manifold defined by the tuple $\gamma(s) = (x^1(s), \ldots, x^n(s))$. Then the tangent to $\gamma(s)$ is given by

$$\frac{d\gamma}{ds} \equiv \frac{dx^i}{ds} \boldsymbol{e}_i \tag{2.130}$$

and if $v(x)$ is a vector field on the manifold then

$$\begin{aligned}
\left(\frac{d\gamma}{ds} \cdot \nabla_x \right) v =& \frac{dx^i}{ds} \frac{\partial}{\partial x^i} \left(v^j \boldsymbol{e}_j \right) \\
=& \frac{dx^i}{ds} \left(\frac{\partial v^j}{\partial x^i} \boldsymbol{e}_j + v^j \frac{\partial \boldsymbol{e}_j}{\partial x^i} \right)
\end{aligned}$$

$$
\begin{aligned}
&= \frac{dx^i}{ds}\left(\frac{\partial v^j}{\partial x^i}\boldsymbol{e}_j + v^j\Gamma^k_{ij}\boldsymbol{e}_k\right)\\
&= \frac{dx^i}{ds}\frac{\partial v^j}{\partial x^i}\boldsymbol{e}_j + \frac{dx^i}{ds}v^k\Gamma^j_{ik}\boldsymbol{e}_j\\
&= \left(\frac{dv^j}{ds} + \frac{dx^i}{ds}v^k\Gamma^j_{ik}\right)\boldsymbol{e}_j\\
&= 0. \qquad (2.131)
\end{aligned}
$$

Thus eq (2.131) is equivalent to eq (2.129) and parallel transport of a vector field along a curve is equivalent to the directional derivative of the vector field in the direction of the tangent to the curve being zero.

If the tensor component representation is contra-variant (superscripts instead of subscripts) we must use the covariant component representation of the vector arguments of the tensor, $a = a_i\boldsymbol{e}^i$. Then the definition of parallel transport gives

$$
\begin{aligned}
(a\cdot\nabla_x)\, b &= a^i\partial_{x^i}\left(b_j\boldsymbol{e}^j\right)\\
&= a^i\left(\left(\partial_{x^i}b_j\right)\boldsymbol{e}^j + b_j\partial_{x^i}\boldsymbol{e}^j\right), \qquad (2.132)
\end{aligned}
$$

and we need

$$
\left(\partial_{x^i}b_j\right)\boldsymbol{e}^j + b_j\partial_{x^i}\boldsymbol{e}^j = 0. \qquad (2.133)
$$

To satisfy equation (2.133) consider the following

$$
\begin{aligned}
\partial_{x^i}\left(\boldsymbol{e}^j\cdot\boldsymbol{e}_k\right) &= 0\\
\left(\partial_{x^i}\boldsymbol{e}^j\right)\cdot\boldsymbol{e}_k + \boldsymbol{e}^j\cdot\left(\partial_{x^i}\boldsymbol{e}_k\right) &= 0\\
\left(\partial_{x^i}\boldsymbol{e}^j\right)\cdot\boldsymbol{e}_k + \boldsymbol{e}^j\cdot\boldsymbol{e}_l\Gamma^l_{ik} &= 0\\
\left(\partial_{x^i}\boldsymbol{e}^j\right)\cdot\boldsymbol{e}_k + \delta^j_l\Gamma^l_{ik} &= 0\\
\left(\partial_{x^i}\boldsymbol{e}^j\right)\cdot\boldsymbol{e}_k + \Gamma^j_{ik} &= 0\\
\left(\partial_{x^i}\boldsymbol{e}^j\right)\cdot\boldsymbol{e}_k &= -\Gamma^j_{ik} \qquad (2.134)
\end{aligned}
$$

Now dot eq (2.133) into $\boldsymbol{e}_k$ giving

$$
\begin{aligned}
\left(\partial_{x^i}b_j\right)\boldsymbol{e}^j\cdot\boldsymbol{e}_k + b_j\left(\partial_{x^i}\boldsymbol{e}^j\right)\cdot\boldsymbol{e}_k &= 0\\
\left(\partial_{x^i}b_j\right)\delta^k_j - b_j\Gamma^j_{ik} &= 0\\
\left(\partial_{x^i}b_k\right) &= b_j\Gamma^j_{ik}. \qquad (2.135)
\end{aligned}
$$

Thus if we have a mixed representation of a tensor

$$
T\left(a_1,\ldots,a_r;x\right) = T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_r}\left(x\right)a^{i_1}\ldots a^{i_s}a_{i_{s+1}}\ldots a_{i_r}, \qquad (2.136)
$$

the covariant derivative of the tensor is

$$\begin{aligned}
(a_{r+1}\cdot D)\,T\,(a_1,\ldots,a_r;x) =& \frac{\partial T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_r}}{\partial x^{r+1}} a^{i_1}\ldots a^{i_s} a_{i_{s+1}}\ldots a^r_{i_r} a^{i_{r+1}} \\
&+ \sum_{p=1}^{s} \frac{\partial a^{i_p}}{\partial x^{i_{r+1}}} T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_r} a^{i_1}\ldots \breve{a}^{i_p}\ldots a^{i_s} a_{i_{s+1}}\ldots a_{i_r} a^{i_{r+1}} \\
&+ \sum_{q=s+1}^{r} \frac{\partial a_{i_p}}{\partial x^{i_{r+1}}} T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_r} a^{i_1}\ldots a^{i_s} a_{i_{s+1}}\ldots \breve{a}_{i_q}\ldots a_{i_r} a^{i_{r+1}} \\
=& \frac{\partial T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_r}}{\partial x^{r+1}} a^{i_1}\ldots a^{i_s} a_{i_{s+1}}\ldots a^r_{i_r} a^{i_{r+1}} \\
&- \sum_{p=1}^{s} \Gamma^{i_p}_{i_{r+1} l_p} T_{i_1\ldots i_p\ldots i_s}{}^{i_{s+1}\ldots i_r} a^{i_1}\ldots a^{l_p}\ldots a^{i_s} a_{i_{s+1}}\ldots a_{i_r} a^{i_{r+1}} \\
&+ \sum_{q=s+1}^{r} \Gamma^{l_q}_{i_{r+1} i_q} T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_q\ldots i_r} a^{i_1}\ldots a^{i_s} a_{i_{s+1}}\ldots a_{l_q}\ldots a_{i_r} a^{i_{r+1}}. \qquad (2.137)
\end{aligned}$$

From eq (2.137) we obtain the components of the covariant derivative to be

$$\frac{\partial T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_r}}{\partial x^{r+1}} - \sum_{p=1}^{s} \Gamma^{i_p}_{i_{r+1} l_p} T_{i_1\ldots i_p\ldots i_s}{}^{i_{s+1}\ldots i_r} + \sum_{q=s+1}^{r} \Gamma^{l_q}_{i_{r+1} i_q} T_{i_1\ldots i_s}{}^{i_{s+1}\ldots i_q\ldots i_r}. \qquad (2.138)$$

The component free form of the covariant derivative (the one used to calculate it in the code) is

$$\mathcal{D}_{a_{r+1}} T\,(a_1,\ldots,a_r;x) \equiv \nabla T - \sum_{k=1}^{r} T\,(a_1,\ldots,(a_{r+1}\cdot\nabla)\,a_k,\ldots,a_r;x)\,. \qquad (2.139)$$

Chapter 3

Module Components

The geometric algebra module consists of the following files and classes

File	Classes	Usage
`metric.py`	`Metric`	Instantiates metric tensor and derivatives of basis vectors. Normalized basis if required.
`ga.py`	`Ga`	Instantiates geometric algebra (inherits `Metric`), generates bases, blades, multiplication tables, reciprocal basis, and left and right geometric derivative operators.
	`Sm`	Instantiates geometric algebra for submainfold (inherits `Ga`).
`mv.py`	`Mv`	Instantiates multivector.
	`Dop`	Instantiates linear multivector differential operator.
`lt.py`	`Lt`	Instantiates multivector linear transformation.
`printer.py`	`Eprint`	Starts enhanced text printing on ANSI terminal (requires `ConEmu` on Windows).
	`GaPrinter`	Text printer for all geometric algebra classes (inherits from `sympy StringPrinter`).
	`GaLatexPrinter`	LaTeXprinter for all geometric algebra classes (inherits from `sympy LatexPrinter`).

3.1 Instantiating a Geometric Algebra

The geometric algebra class is instantiated with

```
Ga(basis, g=None, coords=None, X=None, norm=False, debug=False)
```

The `basis` and `g` parameters were described in section 2.2. If the metric is a function of position, if we have multivector fields, or we wish to calculate geometric derivatives a coordinate set, `coords`, is required. `coords` is a list of *sympy* symbols. For the case of instantiating a 3-d geometric algebra in spherical coordinates we have

```
(r, th, phi) = coords = symbols('r,theta,phi', real=True)
basis = 'e_r e_theta e_phi'
g = [1, r**2, r**2*sin(th)**2]
sp3d = Ga(basis,g=g,coords=coords,norm=True)
```

The input `X` allows the metric to be input as a vector manifold. `X` is a list of functions of `coords` of dimension, m, equal to or greater than the number of coordinates. If `g=None` it is assumed that `X` is a vector in an m-dimensional orthonormal Euclidean vector space. If it is wished the embedding vector space to be non-Euclidean that condition is specified with `g`. For example if we wish the embedding space to be a 5-dimensional Minkowski space then `g=[-1,1,1,1,1]`. Then the Ga class uses `X` to calculate the manifold basis vectors as a function of the coordinates and from them the metric tensor.[1]

If `norm=True` the basis vectors of the manifold are normalized so that the absolute values of the squares of the basis vectors are one. *Currently you only use this option for diagonal metric tensors, and even there due so with caution, due to the possible problems with taking the square root of a general* sympy *expression (one that has an unknown sign).*

If `debug=True` the data structures required to initialize the Ga class are printed out.

To get the basis vectors for `sp3d` we would have to use the member function `Ga.mv()` in the form

```
(er,eth,ephi) = sp3d.mv()
```

[1]Since `X` or the metric tensor can be functions of coordinates the vector space that the geometric algebra is constructed from is not necessarily flat so that the geometric algebra is actually constructed on the tangent space of the manifold which is a vector space.

To access the reciprocal basis vectors of the geometric algebra use the member function `mvr()`

`Ga.mvr(norm='True')`

> `Ga.mvr(norm)` returns the reciprocal basis vectors as a tuple. This allows the programmer to attach any python variable names to the reciprocal basis vectors that is convenient. For example (demonstrating the use of both `mv()` and `mvr(norm)`)
>
> ```
> (e_x,e_y,e_z) = o3d.mv()
> (e__x,e__y,e__z) = o3d.mvr()
> ```
>
> If `norm='True'` or the basis vectors are orthogonal the dot product of the basis vector and the corresponding reciprocal basis vector is one $\left(e_i \cdot e^j = \delta_i^j\right)$. If `norm='False'` and the basis is non-orthogonal The dot product of the basis vector and the corresponding reciprocal basis vector is the square of the pseudo scalar, I^2, of the geometric algebra $\left(e_i \cdot e^j = I^2\delta_i^j\right)$.

In addition to the basis vectors, if coordinates are defined for the geometric algebra, the left and right geometric derivative operators are calculated and accessed with the `Ga` member function `grads()`.

`Ga.grads()`

> `Ga.grads()` returns a tuple with the left and right geometric derivative operators. A typical usage would be
>
> ```
> (grad,rgrad) = sp3d.grads()
> ```
>
> for the spherical 3-d geometric algebra. The left derivative (`grad` $= \boldsymbol{\nabla}$) and the right derivative $\left(\texttt{rgrad} = \bar{\boldsymbol{\nabla}}\right)$ have been explained in section 2.10.3. Again the names `grad` and `rgrad` used in a program are whatever the user chooses them to be. In the previous example `grad` and `rgrad`.

an alternative instantiation method is

`Ga.build(basis, g=None, coords=None, X=None, norm=False, debug=False)`

> The input parameters for `Ga.build()` are the same as for `Ga()`. The difference is that in addition to returning the geometric algebra `Ga.build()` returns the basis vectors at the same time. Using `Ga.build()` in the previous example gives
>
> ```
> (r, th, phi) = coords = symbols('r,theta,phi', real=True)
> basis = 'e_r e_theta e_phi'
> ```

```
g = [1, r**2, r**2*sin(th)**2]
(sp3d,er,eth,ephi) = Ga.build(basis,g=g,coord=coords,norm=True)
```

To access the pseudo scalar of the geometric algebra us the member function `I()`.

`Ga.I()`

> `Ga.I()` returns the normalized pseudo scalar ($|I^2| = 1$) for the geometric algebra. For example $I =$ `o3d.I()` for the `o3d` geometric algebra.

In general we have defined member functions of the `Ga` class that will instantiate objects of other classes since the objects of the other classes are all associated with a particular geometric algebra object. Thus we have

Object	Class	`Ga` method
multivector	`Mv`	`mv`
submanifold	`Sm`	`sm`
linear transformation	`Lt`	`lt`
differential operator	`Dop`	`dop`

for the instantiation of various objects from the `Ga` class. This means that in order to instantiate any of these objects we need only to import `Ga` into our program.

3.2 Instantiating a Multivector

Since we need to associate each multivector with the geometric algebra that contains it we use a member function of Ga to instantiate every multivector[2] The multivector is instantiated with:

`Ga.mv(name, mode, f=False)`

> As an example of both instantiating a geometric algebra and multivectors consider the following code fragment for a 3-d Euclidean geometric algebra.

```
from sympy import symbols
from ga import Ga
(x, y, z) = coords = symbols('x,y,z',real=True)
o3d = Ga('e_x e_y e_z', g=[1,1,1], coords=coords)
```

[2]There is a multivector class, `Mv`, but in order the insure that every multivector is associated with the correct geometric algebra we always use the member function `Ga.mv` to instantiate the multivector.

```
5 (ex, ey, ez) = o3d.mv()
6 V = o3d.mv('V','vector',f=True)
7 f = o3d.mv(x*y*z)
8 B = o3d.mv('B',2)
```

First consider the multivector instantiation in line 6,

```
V = o3d.mv('V','vector',f=True)
```

.

Here a 3-dimensional multivector field that is a function of `x`, `y`, and `z` (`f=True`) is being instantiated. If latex output were used (to be discussed later) the multivector `V` would be displayed as

$$V^x \boldsymbol{e}_x + V^y \boldsymbol{e}_y + V^z \boldsymbol{e}_z \tag{3.1}$$

Where the coefficients of the basis vectors are generalized *sympy* functions of the coordinates. The superscripts[3] are formed from the coordinate symbols or if there are no coordinates from the subscripts of the basis vectors. The types of name and modes available for multivector instantiation are

`name`	`mode`	result
string s	`'scalar'`	symbolic scalar of value Symbol(s)
string s	`'vector'`	symbolic vector
string s	`'grade2'` or `'bivector'`	symbolic bivector
string s	`r` (integer)	symbolic r-grade multivector
string s	`'pseudo'`	symbolic pseudoscalar
string s	`'spinor'`	symbolic even multivector
string s	`'mv'`	symbolic general multivector
scalar c	None	zero grade multivector with coefficient value c

Line 5 of the previous listing illustrates the case of using the `mv` member function with no arguments. The code does not return a multivector, but rather a tuple or the basis vectors of the geometric algebra `o3d`. The elements of the tuple then can be used to construct multivectors, or multivector fields through the operations of addition, subtraction, multiplication (geometric, inner, and outer products and left and right contraction). As an example we could construct the vector function

```
F = x**2*ex + z*ey + x*y*ez
```

or the bivector function

```
B = z*(ex^ey) + y*(ey^ez) + y*(ex^ez).
```

[3]Denoted in text output by `A__x`, etc. so that for text output `A` would be printed as `A__x*e_x+A__y*e_y+A__z*e_z`.

Line 7 is an example of instantiating a multivector scalar function (a multivector with only a scalar part). If we print `f` the result is `x*y*z`. Line 8 is an example of instantiating a grade r (in the example a grade 2) multivector where

$$B = B^{xy}\boldsymbol{e}_x \wedge \boldsymbol{e}_y + B^{yz}\boldsymbol{e}_y \wedge \boldsymbol{e}_z + B^{xz}\boldsymbol{e}_x \wedge \boldsymbol{e}_z. \tag{3.2}$$

If one wished to calculate the left and right geometric derivatives of `F` and `B` the required code would be

```
1 (grad,rgrad) = o3d.grads()
2 dF = grad*F
3 dB = grad*B
4 dFr = F*rgrad
5 dBr = B*rgrad.
```

`dF`, `dB`, `dFr`, and `dBr` are all multivector functions. For the code where the order of the operations are reversed

```
1 (grad,rgrad) = o3d.grads()
2 dFop = F*grad
3 dBop = B*grad
4 dFrop = rgrad*F
5 dBrop = rgrad*B.
```

`dFop`, `dBop`, `dFrop`, and `dBrop` are all multivector differential operators (again see section 2.10.3).

3.3 Backward Compatibility Class MV

In order to be backward compatible with older versions of *galgebra* we introduce the class MV which is inherits it's functions from then class Mv. To instantiate a geometric algebra using MV use the static function

```
1 MV.setup(basis, metric=None, coords=None, rframe=False,\
2 debug=False,curv=(None,None))}
```

This function allows a single geometric algebra to be created. If the function is called more than once the old geometric algebra is overwritten by the new geometric algebra. The named input `metric` is the same as the named input `g` in the current

version of *galgebra*. Likewise, `basis`, `coords`, and `debug` are the same in the old and current versions of *galgebra*[4]. Due to improvements in *sympy* the inputs `rframe` and `curv[1]` are no longer required. `curv[0]` is the vector function (list or tuple of scalar functions) of the coordinates required to define a vector manifold. For compatibility with the old version of *galgebra* if `curv` is used `metric` should be a orthonormal Euclidean metric of the same dimension as `curv[0]`. It is strongly suggested that one use the new methods of defining a geometric algebra on a manifold.

`MV(base, mvtype, fct=False, blade_rep=True)`

For the instantiation of multivector using `MV` the `base` and `mvtype` arguments are the same as for new methods of multivector instantiation. The `fct` input is the same and the `g` input in the new methods. `blade_rep` is not used in the new methods so setting `blade_rep=False` will do nothing. Effectively `blade_rep=False` was not used in the old examples.

`Fmt(self, fmt=1, title=None)`

`Fmt` in `MV` has inputs identical to `Fmt` in `Mv` except that if `A` is a multivector then `A.Fmt(2,'A')` executes a print statement from `MV` and returns `None`, while from `Mv`, `A.Fmt(2,'A')` returns a string.

3.4 Basic Multivector Class Functions

If we can instantiate multivectors we can use all the multivector class functions as described as follows.

`convert_to_blades(self)`

Convert multivector from the base representation to the blade representation. If multivector is already in blade representation nothing is done.

`convert_from_blades(self)`

Convert multivector from the blade representation to the base representation. If multivector is already in base representation nothing is done.

[4]If the metric is input as a list or list or lists the object is no longer quoted (input as a string). For example the old `metric='[1,1,1]'` becomes `metric=[1,1,1]`.

`diff(self,var)`

Calculate derivative of each multivector coefficient with respect to variable `var` and form new multivector from coefficients.

`dual(self)`

The mode of the `dual()` function is set by the `Ga` class static member function, `GA.dual_mode(mode='I+')` of the `GA` geometric galgebra which sets the following return values (I is the pseudo-scalar for the geometric algebra `GA`)

`mode`	Return Value
`'+I'`	IA
`'I+'`	AI
`'-I'`	$-IA$
`'I-'`	$-AI$
`'+Iinv'`	$I^{-1}A$
`'Iinv+'`	AI^{-1}
`'-Iinv'`	$-I^{-1}A$
`'Iinv-'`	$-AI^{-1}$

For example if the geometric algebra is `o3d`, `A` is a multivector in `o3d`, and we wish to use `mode='I-'`. We set the mode with the function `o3d.dual('I-')` and get the dual of `A` with the function `A.dual()` which returns $-AI$.

If `o3d.dual(mode)` is not called the default for the dual mode is `mode='I+'` and `A*I` is returned.

`even(self)`

Return the even grade components of the multivector.

`exp(self,hint='-')`

Return exponential of a multivector A if A^2 is a scalar (if A^2 is not a scalar an error message is generated). If A is the multivector then e^A is returned where the default `hint='-'`, assumes $A^2 < 0$ so that

$$e^A = \cos\left(\sqrt{-A^2}\right) + \sin\left(\sqrt{-A^2}\right)\frac{A}{\sqrt{-A^2}}.$$

If the hint is not `-` then $A^2 > 0$ is assumed so that

$$e^A = \cosh\left(\sqrt{A^2}\right) + \sinh\left(\sqrt{A^2}\right)\frac{A}{\sqrt{A^2}}.$$

The hint is required for symbolic multivectors A since in general *sympy* cannot determine if A^2 is positive or negative. If A is purely numeric the hint is ignored.

`expand(self)`

Return multivector in which each coefficient has been expanded using *sympy* `expand()` function.

`factor(self)`

Apply the `sympy` `factor` function to each coefficient of the multivector.

`Fmt(self, fmt=1,title=None)`

Fuction to print multivectors in different formats where

`fmt`	Formatting
1	Print entire multivector on one line.
2	Print each grade of multivector on one line.
3	Print each base of multivector on one line.

`title` appends a title string to the beginning of the output. An equal sign in the title string is not required, but is added as a default.

`func(self,fct)`

Apply the `sympy` scalar function `fct` to each coefficient of the multivector.

`grade(self,igrade=0)`

Return a multivector that consists of the part of the multivector of grade equal to `igrade`. If the multivector has no `igrade` part return a zero multivector.

`inv(self)`

Return the inverse of the multivector M (`M.inv()`) if $MM^\dagger$ is a non-zero scalar. If $MM^\dagger$ is not a scalar the program exits with an error message.

`norm(self,hint='+')`

Return the norm of the multivector M (`M.norm()`) defined by $\sqrt{|MM^\dagger|}$. If $MM^\dagger$ is a scalar (a *sympy* scalar is returned). If $MM^\dagger$ is not a scalar the program exits with an error message. If $MM^\dagger$ is a number *sympy* can determine if it is positive or negative and calculate the absolute value. If $MM^\dagger$ is a *sympy* expression (function) *sympy* cannot determine the sign of the expression so that `hint='+'` or `hint='-'` is needed to determine if the program should calculate $\sqrt{MM^\dagger}$ or $\sqrt{-MM^\dagger}$. For example if we are in a Euclidean space and `M` is a vector then `hint='+'`, if `M` is a bivector then let `hint='-'`. If `hint='0'` and $MM^\dagger$ is a symbolic scalar `sqrt(Abs(M*M.rev()))` is returned where `Abs()` is the *sympy* symbolic absolute value function.

`norm2(self)`

Return the square of the norm of the multivector M (`M.norm2()`) defined by $MM^\dagger$ if $MM^\dagger$ is a scalar (a *sympy* scalar is returned). If $MM^\dagger$ is not a scalar the program exits with an error message.

`project_in_blade(self,blade)`

Return the projection of the mutivector A in subspace defined by the blade, B, using the formula $(A \rfloor B)\, B^{-1}$ in [3], page 121.

`reflect_in_blade(self,blade)`

Return the reflection of the mutivector A in the subspace defined by the r-grade blade, B_r, using the formula (extended to multivectors) $\sum_i (-1)^{r(i+1)} B_r \langle A \rangle_i B_r^{-1}$ in [3], page 129.

`rev(self)`

Return the reverse of the multivector. See eq. (2.47).

`rotate_multivector(self,itheta,hint='-')`

Rotate the multivector A via the operation $e^{-\theta i/2} A e^{\theta i/2}$ where itheta $= \theta i$, θ is a scalar, and i is a unit, $i^2 = \pm 1$, 2-blade. If $(\theta i)^2$ is not a number `hint` is required to determine the sign of the square of `itheta`. The default chosen, `hint='-'`, is correct for any Euclidean space.

`scalar(self)`

Return the coefficient (*sympy* scalar) of the scalar part of a multivector.

`simplify(self,mode=simplify)`

`mode` is a *sympy* simplification function of a list/tuple of *sympy* simplification functions that are applied in sequence (if more than one function) each coefficient of the multivector. For example if we wished to applied `trigsimp` and `ratsimp` *sympy* functions to the multivector `F` the code would be

```
Fsimp = F.simplify(mode=[trigsimp,ratsimp]).
```

Actually `simplify` could be used to apply any scalar *sympy* function to the coefficients of the multivector.

`set_coef(self,grade,base,value)`

Set the multivector coefficient of index `(grade,base)` to `value`.

`subs(self,x)`

Return multivector where *sympy* subs function has been applied to each coefficient of multivector for argument dictionary/list `x`.

`trigsimp(self,**kwargs)`

Apply the `sympy` trigonometric simplification function `trigsimp` to each coefficient of the multivector. `**kwargs` are the arguments of trigsimp. See `sympy` documentation on `trigsimp` for more information.

3.5 Basic Multivector Functions

`com(A,B)`

Calculate commutator of multivectors A and B. Returns $(AB - BA)/2$.

`cross(v1,v2)`

If `v1` and `v2` are 3-dimensional Euclidean vectors the vector cross product is returned, $v_1 \times v_2 = -I\,(v_1 \wedge v_2)$.

`def_prec(gd,op_ord='<>|,^,*')`[5]

[5] See footnote 6.

This is used with the `GAeval()` function to evaluate a string representing a multivector expression with a revised operator precedence. `def_prec()` redefines the operator precedence for multivectors. `def_prec()` must be called in the main program an the argument `gd` must be `globals()`. The argument `op_ord` defines the order of operator precedence from high to low with groups of equal precedence separated by commas. the default precedence `op_ord='<>|,^,\*'` is that used by Hestenes ([2],p7,[1],p38).

`dual(A,mode='I+')`

Return the dual of the multivector `A`. The default operation is AI. For other modes see member function`Mv.dual(mode)`

`exp(A,hint='-')`

If A is a multivector then `A.exp(hint)` is returned. If A is a *sympy* expression the *sympy* expression e^A is returned (see `sympy.exp(A)` member function).

`GAeval(s,pstr=False)`[6]

Returns multivector expression for string `s` with operator precedence for string `s` defined by inputs to function `def_prec()`. if `pstr=True` `s` and `s` with parenthesis added to enforce operator precedence are printed.

`grade(A,r=0)`

If A is a multivector $\langle A \rangle_r$ is returned.

`inv(A)`

If A is a multivector and $AA^\dagger$ is a non-zero scalar then $A^{-1} = A^\dagger/(AA^\dagger)$ is returned otherwise an exception is returned.

`Nga(x,prec=5)`

If `x` is a multivector with coefficients that contain floating point numbers, `Nga()` rounds all these numbers to a precision of `prec` and returns the rounded multivector.

`norm(A,hint='-')`

If A is a multivector and $AA^\dagger$ is a number (not a scalar function) then $\sqrt{|AA^\dagger|}$ is returned. If $AA^\dagger$ is a scalar *sympy* expression, but not a number, and `hint='-'` then

[6] `GAeval` is in the `printer` module.

return $\sqrt{-AA^\dagger}$ otherwise return $\sqrt{AA^\dagger}$.

`norm2(A)`

If A is a multivector and $AA^\dagger$ is a scalar return $\left|AA^\dagger\right|$.

`proj(B,A)`

Project blade `A` on blade `B` returning $(A \rfloor B)\, B^{-1}$.

`ReciprocalFrame(basis,mode='norm')`

If `basis` is a list/tuple of vectors, `ReciprocalFrame()` returns a tuple of reciprocal vectors. If `mode=norm` the vectors are normalized. If `mode` is anything other than `norm` the vectors are unnormalized and the normalization coefficient is added to the end of the tuple. One must divide by this coefficient to normalize the vectors.

`refl(B,A)`

Reflect multivector A in blade B. If s is grade of B returns $\sum_r (-1)^{s(r+1)} B \left\langle A \right\rangle_r B^{-1}$.

`rev(A)`

If A is a multivector return $A^\dagger$.

`rot(itheta,A,hint='-')`

If `A` is a multivector return `A.rotate_multivector(itheta,hint)` where `itheta` is the bi-vector blade defining the rotation. For the use of `hint` see the member function `Mv.rotate_multivector(self,itheta,hint)`.

3.6 Multivector Derivatives

The various derivatives of a multivector function is accomplished by multiplying the gradient operator vector with the function. The gradient operation vector is returned by the `Ga.grads()` function if coordinates are defined. For example if we have for a 3-D vector space

```
1  X = (x,y,z) = symbols('x y z')
2  o3d = Ga('e*x|y|z',metric='[1,1,1]',coords=X)
3  (ex,ey,ez) = o3d.mv()
4  (grad,rgrad) = o3d.grads()
```

Then the gradient operator vector is `grad` (actually the user can give it any name he wants to). The derivatives of the multivector function `F = o3d.mv('F','mv',f=True)` are given by multiplying by the left geometric derivative operator and the right geometric derivative operator ($\texttt{grad} = \nabla$ and $\texttt{rgrad} = \bar{\nabla}$). Another option is to use the radiant operator members of the geometric algebra directly where we have $\nabla = \texttt{o3d.grad}$ and $\bar{\nabla} = \texttt{o3d.rgrad}$.

$$
\begin{aligned}
\nabla F &= \texttt{grad*F} \\
F\bar{\nabla} &= \texttt{F*rgrad} \\
\nabla \wedge F &= \texttt{grad\^{}F} \\
F \wedge \bar{\nabla} &= \texttt{F\^{}rgrad} \\
\nabla \cdot F &= \texttt{grad|F} \\
F \cdot \bar{\nabla} &= \texttt{F|rgrad} \\
\nabla \lfloor F &= \texttt{grad<F} \\
F \lfloor \bar{\nabla} &= \texttt{F<rgrad} \\
\nabla \rfloor F &= \texttt{grad>F} \\
F \rfloor \bar{\nabla} &= \texttt{F>rgrad}
\end{aligned}
$$

The preceding list gives examples of all possible multivector derivatives of the multivector function `F` where the operation returns a multivector function. The complementary operations

$$
\begin{aligned}
F\nabla &= \texttt{F*grad} \\
\bar{\nabla}F &= \texttt{rgrad*F} \\
F \wedge \nabla &= \texttt{F\^{}grad} \\
\bar{\nabla} \wedge F &= \texttt{rgrad\^{}F} \\
F \cdot \nabla &= \texttt{F|grad} \\
\bar{\nabla} \cdot F &= \texttt{rgrad|F} \\
F \lfloor \nabla &= \texttt{F<grad} \\
\bar{\nabla} \lfloor F &= \texttt{rgrad<F} \\
F \rfloor \nabla &= \texttt{F>grad} \\
\bar{\nabla} \rfloor F &= \texttt{rgrad>F}
\end{aligned}
$$

all return multivector linear differential operators.

3.7 Submanifolds

In general the geometric algebra that the user defines exists on the tangent space of a manifold (see section 2.9). The submanifold class, `Sm`, is derived from the `Ga` class and allows one to define a submanifold of a manifold by defining a coordinate mapping between the submanifold coordinates and the manifold coordinates. What is returned as the submanifold is the geometric algebra of the tangent space of the submanifold. The submanifold for a geometric algebra is instantiated with

`Ga.sm(map,coords,root='e',norm=False)`

> To define the submanifold we must def a coordinate map from the coordinates of the submanifold to each of the coordinates of the base manifold. Thus the arguments `map` and `coords` are respectively lists of functions and symbols. The list of symbols, `coords`, are the coordinates of the submanifold and are of length equal to the dimension of the submanifold. The list of functions, `map`, define the mapping from the coordinate space of the submanifold to the coordinate space of the base manifold. The length of `map` is equal to the dimension of the base manifold and each function in `map` is a function of the coordinates of the submanifold. `root` is the root of the string that is used to name the basis vectors of the submanifold. The default value of `root` is `e`. The result of this is that if the *sympy* symbols for the coordinates are `u` and `v` (two dimensional manifold) the text symbols for the basis vectors are `e_u` and `e_v` or in LaTeX e_u and e_v. As a concrete example consider the following code.

Listing 3.1: python/submanifold.py

```
1 from sympy import symbols, sin, pi, latex
2 from ga import Ga
3 from printer import Format, xpdf
4
5 Format()
6 coords = (r, th, phi) = symbols('r,theta,phi', real=True)
7 sp3d = Ga('e_r e_th e_ph', g=[1, r**2, r**2*sin(th)**2],\
8           coords=coords, norm=True)
9
10 sph_uv = (u, v) = symbols('u,v', real=True)
11 sph_map = [1, u, v]  # Coordinate map for sphere of r = 1
12 sph2d = sp3d.sm(sph_map,sph_uv)
13
14 print r'(u,v)\rightarrow (r,\theta,\phi) = ',latex(sph_map)
```

```
print 'g =',latex(sph2d.g)
F = sph2d.mv('F','vector',f=True) #scalar function
f = sph2d.mv('f','scalar',f=True) #vector function
print r'\nabla f =',sph2d.grad * f
print 'F =',F
print r'\nabla F = ',sph2d.grad * F

cir_s = s = symbols('s',real=True)
cir_map = [pi/8,s]
cir1d = sph2d.sm(cir_map,(cir_s,))

print 'g =',latex(cir1d.g)
h = cir1d.mv('h','scalar',f=True)
H = cir1d.mv('H','vector',f=True)
print r'(s)\rightarrow (u,v) = ',latex(cir_map)
print 'H =', H
print latex(H)
print r'\nabla h =', cir1d.grad * h
print r'\nabla H =', cir1d.grad * H
xpdf(filename='submanifold.tex',paper=(6,5),crop=True)
```

The output of this program (using LaTeX) is

$$(u,v) \rightarrow (r,\theta,\phi) = [1, \quad u, \quad v]$$

$$g = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\nabla f = \partial_u f \boldsymbol{e}_u + \partial_v f \boldsymbol{e}_v$$

$$F = F^u \boldsymbol{e}_u + F^v \boldsymbol{e}_v$$

$$\nabla F = \left(\partial_u F^u + \partial_v F^v\right) + \left(-\partial_v F^u + \partial_u F^v\right) \boldsymbol{e}_u \wedge \boldsymbol{e}_v$$

$$g = \left[1\right]$$

$$(s) \rightarrow (u,v) = \left[\frac{\pi}{8}, \quad s\right]$$

$$H = H^s \boldsymbol{e}_s$$

$$H^s \boldsymbol{e}_s$$

$$\nabla h = \partial_s h \boldsymbol{e}_s$$

$$\nabla H = \partial_s H^s$$

The base manifold, `sp3d`, is a 3-d Euclidean space using standard spherical coordi-

nates. The submanifold `sph2d` of `sp3d` is a spherical surface of radius 1. To take the sumanifold operation one step further the submanifold `cir1d` of `sph2d` is a circle in `sph2d` where the latitude of the circle is $\pi/8$.

In each case, for demonstration purposes, a scalar and vector function on each manifold is defined (`f` and `F` for the 2-d manifold and `h` and `H` for the 1-d manifold) and the geometric derivative of each function is taken. The manifold mapping and the metric tensor for `cir1d` of `sph2d` are also shown. Note that if the submanifold basis vectors are not normalized[7] the program output is.

$$(u, v) \to (r, \theta, \phi) = \begin{bmatrix} 1, & u, & v \end{bmatrix}$$

$$g = \begin{bmatrix} 1 & 0 \\ 0 & \sin^2(u) \end{bmatrix}$$

$$\nabla f = \partial_u f \boldsymbol{e_u} + \frac{\partial_v f}{\sin^2(u)} \boldsymbol{e_v}$$

$$F = F^u \boldsymbol{e_u} + F^v \boldsymbol{e_v}$$

$$\nabla F = \left(\frac{F^u}{\tan(u)} + \partial_u F^u + \partial_v F^v \right) + \left(\frac{2F^v}{\tan(u)} + \partial_u F^v - \frac{\partial_v F^u}{\sin^2(u)} \right) \boldsymbol{e_u} \wedge \boldsymbol{e_v}$$

$$g = \begin{bmatrix} -\frac{\sqrt{2}}{4} + \frac{1}{2} \end{bmatrix}$$

$$(s) \to (u, v) = \begin{bmatrix} \frac{\pi}{8}, & s \end{bmatrix}$$

$$H = H^s \boldsymbol{e_s}$$

$$H^s \boldsymbol{e_s}$$

$$\nabla h = \left(2\sqrt{2} + 4 \right) \partial_s h \boldsymbol{e_s}$$

$$\nabla H = \partial_s H^s$$

3.8 Linear Transformations

The mathematical background for linear transformations is in section 2.11. Linear transformations on the tangent space of the manifold are instantiated with the `Ga` member function `lt` (the actual class being instantiated is `Lt`) as shown in lines 12, 20, 26, and 44 of the code listing `Ltrans.py`. In all of the examples in `Ltrans.py` the default instantiation is used which produces a general (all the coefficients of the linear transformation are symbolic constants) linear

[7] Remember that normalization is currently supported only for orthogonal systems (diagonal metric tensors).

transformation. *Note that to instantiate linear transformations coordinates, $\{\boldsymbol{e}_i\}$, must be defined when the geometric algebra associated with the linear transformation is instantiated. This is due to the naming conventions of the general linear transformation (coordinate names are used) and for the calculation of the trace of the linear transformation which requires taking a divergence.*.To instantiate a specific linear transformation the usage of `lt()` is `Ga.lt(M,f=False)`

`M` is an expression that can define the coefficients of the linear transformation in various ways defined as follows.

`M`	Result
string `M`	Coefficients are symbolic constants with names $\mathtt{M}^{x_i x_j}$ where x_i and x_j are the names of the i^{th} and j^{th} coordinates (see output of `Ltrans.py`).
list `M`	If `M` is a list of multivectors equal in length to the dimension of the vector space then the linear transformation is $L(\boldsymbol{e}_i) = \mathtt{M}[i]$. If `M` is a list of lists of scalars where all lists are equal in length to the dimension of the vector space then the linear transformation is $L(\boldsymbol{e}_i) = \mathtt{M}[i][j]\,\boldsymbol{e}_j$.
dict `M`	If `M` is a dictionary the linear transformation is defined by $L(\boldsymbol{e}_i) = \mathtt{M}[\boldsymbol{e}_i]$. If $\boldsymbol{e}_i$ is not in the dictionary then $L(\boldsymbol{e}_i) = 0$.
rotor `M`	If `M` is a rotor, $\mathtt{M}\mathtt{M}^\dagger = 1$, the linear transformation is defined by $L(\boldsymbol{e}_i) = \mathtt{M}\boldsymbol{e}_i\mathtt{M}^\dagger$.
multivector function `M`	If `M` is a general multivector function, the function is tested for linearity, and if linear the coefficients of the linear transformation are calculated from $L(\boldsymbol{e}_i) = \mathtt{M}(\boldsymbol{e}_i)$.

`f` is `True` or `False`. If `True` the symbolic coefficients of the general linear transformation are instantiated as functions of the coordinates.

The different methods of instantiation are demonstrated in the code `LtransInst.py`

Listing 3.2: python/LtransInst.py

```
from sympy import symbols, sin, cos, latex, Matrix
from ga import Ga
from printer import Format, xpdf

Format()
(x, y, z) = xyz = symbols('x,y,z',real=True)
(o3d, ex, ey, ez) = Ga.build('e_x e_y e_z', g=[1, 1, 1], coords=xyz)
```

```
8
9  A = o3d.lt('A')
10 print r'\mbox{General Instantiation: }A =', A
11 th = symbols('theta',real=True)
12 R = cos(th/2)+(ex^ey)*sin(th/2)
13 B = o3d.lt(R)
14 print r'\mbox{Rotor: }R =', R
15 print r'\mbox{Rotor Instantiation: }B =', B
16 dict1 = {ex:ey+ez,ez:ey+ez,ey:ex+ez}
17 C = o3d.lt(dict1)
18 print r'\mbox{Dictionary} =', latex(dict1)
19 print r'\mbox{Dictionary Instantiation: }C =', C
20 lst1 = [[1,0,1],[0,1,0],[1,0,1]]
21 D = o3d.lt(lst1)
22 print r'\mbox{List} =', latex(lst1)
23 print r'\mbox{List Instantiation: }D =', D
24 lst2 = [ey+ez,ex+ez,ex+ey]
25 E = o3d.lt(lst2)
26 print r'\mbox{List} =', latex(lst2)
27 print r'\mbox{List Instantiation: }E =', E
28 xpdf(paper=(10,12),crop=True)
```

with output

$$\text{General Instantiation: } A = \begin{bmatrix} L(\boldsymbol{e_x}) = & A_{xx}\boldsymbol{e_x} + A_{yx}\boldsymbol{e_y} + A_{zx}\boldsymbol{e_z} \\ L(\boldsymbol{e_y}) = & A_{xy}\boldsymbol{e_x} + A_{yy}\boldsymbol{e_y} + A_{zy}\boldsymbol{e_z} \\ L(\boldsymbol{e_z}) = & A_{xz}\boldsymbol{e_x} + A_{yz}\boldsymbol{e_y} + A_{zz}\boldsymbol{e_z} \end{bmatrix}$$

$$\text{Rotor: } R = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\boldsymbol{e_x} \wedge \boldsymbol{e_y}$$

$$\text{Rotor Instantiation: } B = \begin{bmatrix} L(\boldsymbol{e_x}) = & \cos(\theta)\boldsymbol{e_x} - \sin(\theta)\boldsymbol{e_y} \\ L(\boldsymbol{e_y}) = & \sin(\theta)\boldsymbol{e_x} + \cos(\theta)\boldsymbol{e_y} \\ L(\boldsymbol{e_z}) = & \boldsymbol{e_z} \end{bmatrix}$$

$$\text{Dictionary} = \left\{\boldsymbol{e_x} : \boldsymbol{e_y} + \boldsymbol{e_z},\quad \boldsymbol{e_y} : \boldsymbol{e_x} + \boldsymbol{e_z},\quad \boldsymbol{e_z} : \boldsymbol{e_y} + \boldsymbol{e_z}\right\}$$

$$\text{Dictionary Instantiation: } C = \begin{bmatrix} L(\boldsymbol{e_x}) = & 0 \\ L(\boldsymbol{e_y}) = & 0 \\ L(\boldsymbol{e_z}) = & 0 \end{bmatrix}$$

$$\text{List} = \left[\left[1,\quad 0,\quad 1\right],\quad \left[0,\quad 1,\quad 0\right],\quad \left[1,\quad 0,\quad 1\right]\right]$$

$$\text{List Instantiation: } D = \begin{bmatrix} L(\boldsymbol{e_x}) = & \boldsymbol{e_x} + \boldsymbol{e_z} \\ L(\boldsymbol{e_y}) = & \boldsymbol{e_y} \\ L(\boldsymbol{e_z}) = & \boldsymbol{e_x} + \boldsymbol{e_z} \end{bmatrix}$$

$$\text{List} = \left[\boldsymbol{e_y} + \boldsymbol{e_z},\quad \boldsymbol{e_x} + \boldsymbol{e_z},\quad \boldsymbol{e_x} + \boldsymbol{e_y}\right]$$

$$\text{List Instantiation: } E = \begin{bmatrix} L(\boldsymbol{e_x}) = & \boldsymbol{e_y} + \boldsymbol{e_z} \\ L(\boldsymbol{e_y}) = & \boldsymbol{e_x} + \boldsymbol{e_z} \\ L(\boldsymbol{e_z}) = & \boldsymbol{e_x} + \boldsymbol{e_y} \end{bmatrix}$$

The member function of the `Lt` class are

`Lt(A)`

Returns the image of the multivector A under the linear transformation L where $L(A)$ is defined by the linearity of L, the vector values $L(\boldsymbol{e}_i)$, and the definition $L(\boldsymbol{e}_{i_1} \wedge \ldots \wedge \boldsymbol{e}_{i_r}) = L(\boldsymbol{e}_{i_1}) \wedge \ldots \wedge L(\boldsymbol{e}_{i_r})$.

`Lt.det()`

Returns the determinant (a scalar) of the linear transformation, L, defined by $\det(L)\, I = L(I)$.

`Lt.adj()`

Returns the adjoint (a linear transformation) of the linear transformation, L, defined by $a \cdot L(b) = b \cdot \bar{L}(a)$ where a and b are any two vectors in the tangent space and $\bar{L}$ is the adjoint of L.

`Lt.tr()`

Returns the trace (a scalar) of the linear transformation, L, defined by $\mathrm{tr}(L) = \nabla_a \cdot L(a)$ where a is a vector in the tangent space.

`Lt.matrix()`

Returns the matrix representation (*sympy* `Matrix`) of the linear transformation, L, defined by $L(\boldsymbol{e}_i) = L_{ij}\boldsymbol{e}_j$ where L_{ij} is the matrix representation.

The `Ltrans.py` demonstrate the use of the various `Lt` member functions and operators. The operators that can be used with linear transformations are `+`, `-`, and `*`. If A and B are linear transformations, V a multivector, and α a scalar then $(A \pm B)(V) = A(V) \pm B(V)$, $(AB)(V) = A(B(V))$, and $(\alpha A)(V) = \alpha A(V)$.

The `matrix()` member function returns a *sympy* `Matrix` object which can be printed in IPython notebook. To directly print an linear transformation in IPython notebook one must implement (yet to be done) a printing method similar to `mv.Fmt()`.

Note that in `Ltrans.py` lines 30 and 49 are commented out since the latex output of those statements would run off the page. The use can uncomment those statements and run the code in the "LaTeX docs" directory to see the output.

Listing 3.3: python/Ltrans.py

```
1 from sympy import symbols, sin, cos, latex
2 from ga import Ga
3 from printer import Format, xpdf
4
5 Format()
6 (x, y, z) = xyz = symbols('x,y,z',real=True)
7 (o3d, ex, ey, ez) = Ga.build('e_x e_y e_z', g=[1, 1, 1], coords=xyz)
8 grad = o3d.grad
9 (u, v) = uv = symbols('u,v',real=True)
10 (g2d, eu, ev) = Ga.build('e_u e_v', coords=uv)
11 grad_uv = g2d.grad
12 A = o3d.lt('A')
13 print '#3d orthogonal ($A,\\;B$ are linear transformations)'
14 print 'A =', A
15 print r'\f{\operatorname{mat}}{A} =', latex(A.matrix())
16 print '\\f{\\det}{A} =', A.det()
```

```
print '\\overline{A} =', A.adj()
print '\\f{\\Tr}{A} =', A.tr()
print '\\f{A}{e_x^e_y} =', A(ex^ey)
print '\\f{A}{e_x}^\\f{A}{e_y} =', A(ex)^A(ey)
B = o3d.lt('B')
print 'A + B =', A + B
print 'AB =', A * B
print 'A - B =', A - B

print '#2d general ($A,\\;B$ are linear transformations)'
A2d = g2d.lt('A')
print 'A =', A2d
print '\\f{\\det}{A} =', A2d.det()
#A2d.adj().Fmt(4,'\\overline{A}')
print '\\f{\\Tr}{A} =', A2d.tr()
print '\\f{A}{e_u^e_v} =', A2d(eu^ev)
print '\\f{A}{e_u}^\\f{A}{e_v} =', A2d(eu)^A2d(ev)
B2d = g2d.lt('B')
print 'B =', B2d
print 'A + B =', A2d + B2d
print 'AB =', A2d * B2d
print 'A - B =', A2d - B2d
a = g2d.mv('a','vector')
b = g2d.mv('b','vector')
print r'a|\f{\overline{A}}{b}-b|\f{\underline{A}}{a} =',\
     ((a|A2d.adj()(b))-(b|A2d(a))).simplify()

print '#4d Minkowski spaqce (Space Time)'
m4d = Ga('e_t e_x e_y e_z', g=[1, -1, -1, -1],\
         coords=symbols('t,x,y,z',real=True))
T = m4d.lt('T')
print 'g =', m4d.g
print r'\underline{T} =',T
print r'\overline{T} =',T.adj()
#m4d.mv(T.det()).Fmt(4,r'\f{\det}{\underline{T}}')
print r'\f{\mbox{tr}}{\underline{T}} =',T.tr()
a = m4d.mv('a','vector')
b = m4d.mv('b','vector')
```

```
print r'a|\f{\overline{T}}{b}-b|\f{\underline{T}}{a} =',\
      ((a|T.adj()(b))-(b|T(a))).simplify()
xpdf(paper=(10,12),debug=True)
```

The output of this code is.

3d orthogonal ($A,\ B$ are linear transformations)

$$A = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_x}\right) = & A_{xx}\boldsymbol{e_x} + A_{yx}\boldsymbol{e_y} + A_{zx}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_y}\right) = & A_{xy}\boldsymbol{e_x} + A_{yy}\boldsymbol{e_y} + A_{zy}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_z}\right) = & A_{xz}\boldsymbol{e_x} + A_{yz}\boldsymbol{e_y} + A_{zz}\boldsymbol{e_z} \end{array} \right\}$$

$$\operatorname{mat}\left(A\right) = \begin{bmatrix} A_{xx} & A_{xy} & A_{xz} \\ A_{yx} & A_{yy} & A_{yz} \\ A_{zx} & A_{zy} & A_{zz} \end{bmatrix}$$

$$\det\left(A\right) = A_{xx}A_{yy}A_{zz} - A_{xx}A_{yz}A_{zy} - A_{xy}A_{yx}A_{zz} + A_{xy}A_{yz}A_{zx} + A_{xz}A_{yx}A_{zy} - A_{xz}A_{yy}A_{zx}$$

$$\overline{A} = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_x}\right) = & A_{xx}\boldsymbol{e_x} + A_{xy}\boldsymbol{e_y} + A_{xz}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_y}\right) = & A_{yx}\boldsymbol{e_x} + A_{yy}\boldsymbol{e_y} + A_{yz}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_z}\right) = & A_{zx}\boldsymbol{e_x} + A_{zy}\boldsymbol{e_y} + A_{zz}\boldsymbol{e_z} \end{array} \right\}$$

$$\operatorname{Tr}\left(A\right) = A_{xx} + A_{yy} + A_{zz}$$

$$A\left(e_x \wedge e_y\right) = \left(A_{xx}A_{yy} - A_{xy}A_{yx}\right)\boldsymbol{e_x} \wedge \boldsymbol{e_y} + \left(A_{xx}A_{zy} - A_{xy}A_{zx}\right)\boldsymbol{e_x} \wedge \boldsymbol{e_z} + \left(A_{yx}A_{zy} - A_{yy}A_{zx}\right)\boldsymbol{e_y} \wedge \boldsymbol{e_z}$$

$$A\left(e_x\right) \wedge A\left(e_y\right) = \left(A_{xx}A_{yy} - A_{xy}A_{yx}\right)\boldsymbol{e_x} \wedge \boldsymbol{e_y} + \left(A_{xx}A_{zy} - A_{xy}A_{zx}\right)\boldsymbol{e_x} \wedge \boldsymbol{e_z} + \left(A_{yx}A_{zy} - A_{yy}A_{zx}\right)\boldsymbol{e_y} \wedge \boldsymbol{e_z}$$

$$A + B = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_x}\right) = & \left(A_{xx} + B_{xx}\right)\boldsymbol{e_x} + \left(A_{yx} + B_{yx}\right)\boldsymbol{e_y} + \left(A_{zx} + B_{zx}\right)\boldsymbol{e_z} \\ L\left(\boldsymbol{e_y}\right) = & \left(A_{xy} + B_{xy}\right)\boldsymbol{e_x} + \left(A_{yy} + B_{yy}\right)\boldsymbol{e_y} + \left(A_{zy} + B_{zy}\right)\boldsymbol{e_z} \\ L\left(\boldsymbol{e_z}\right) = & \left(A_{xz} + B_{xz}\right)\boldsymbol{e_x} + \left(A_{yz} + B_{yz}\right)\boldsymbol{e_y} + \left(A_{zz} + B_{zz}\right)\boldsymbol{e_z} \end{array} \right\}$$

$$AB = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_x}\right) = & \left(A_{xx}B_{xx} + A_{xy}B_{yx} + A_{xz}B_{zx}\right)\boldsymbol{e_x} + \left(A_{yx}B_{xx} + A_{yy}B_{yx} + A_{yz}B_{zx}\right)\boldsymbol{e_y} + \left(A_{zx}B_{xx} + A_{zy}B_{yx} + A_{zz}B_{zx}\right)\boldsymbol{e_z} \\ L\left(\boldsymbol{e_y}\right) = & \left(A_{xx}B_{xy} + A_{xy}B_{yy} + A_{xz}B_{zy}\right)\boldsymbol{e_x} + \left(A_{yx}B_{xy} + A_{yy}B_{yy} + A_{yz}B_{zy}\right)\boldsymbol{e_y} + \left(A_{zx}B_{xy} + A_{zy}B_{yy} + A_{zz}B_{zy}\right)\boldsymbol{e_z} \\ L\left(\boldsymbol{e_z}\right) = & \left(A_{xx}B_{xz} + A_{xy}B_{yz} + A_{xz}B_{zz}\right)\boldsymbol{e_x} + \left(A_{yx}B_{xz} + A_{yy}B_{yz} + A_{yz}B_{zz}\right)\boldsymbol{e_y} + \left(A_{zx}B_{xz} + A_{zy}B_{yz} + A_{zz}B_{zz}\right)\boldsymbol{e_z} \end{array} \right\}$$

$$A - B = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_x}\right) = & \left(A_{xx} - B_{xx}\right)\boldsymbol{e_x} + \left(A_{yx} - B_{yx}\right)\boldsymbol{e_y} + \left(A_{zx} - B_{zx}\right)\boldsymbol{e_z} \\ L\left(\boldsymbol{e_y}\right) = & \left(A_{xy} - B_{xy}\right)\boldsymbol{e_x} + \left(A_{yy} - B_{yy}\right)\boldsymbol{e_y} + \left(A_{zy} - B_{zy}\right)\boldsymbol{e_z} \\ L\left(\boldsymbol{e_z}\right) = & \left(A_{xz} - B_{xz}\right)\boldsymbol{e_x} + \left(A_{yz} - B_{yz}\right)\boldsymbol{e_y} + \left(A_{zz} - B_{zz}\right)\boldsymbol{e_z} \end{array} \right\}$$

2d general ($A,\ B$ are linear transformations)

$$A = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_u}\right) = & A_{uu}\boldsymbol{e_u} + A_{vu}\boldsymbol{e_v} \\ L\left(\boldsymbol{e_v}\right) = & A_{uv}\boldsymbol{e_u} + A_{vv}\boldsymbol{e_v} \end{array} \right\}$$

$$\det\left(A\right) = A_{uu}A_{vv} - A_{uv}A_{vu}$$

$$\operatorname{Tr}\left(A\right) = \frac{\left(e_u \cdot e_u\right)\left(e_v \cdot e_v\right)A_{uu}}{\left(e_u \cdot e_u\right)\left(e_v \cdot e_v\right) - \left(e_u \cdot e_v\right)^2} + \frac{\left(e_u \cdot e_u\right)\left(e_v \cdot e_v\right)A_{vv}}{\left(e_u \cdot e_u\right)\left(e_v \cdot e_v\right) - \left(e_u \cdot e_v\right)^2} - \frac{\left(e_u \cdot e_v\right)^2 A_{uu}}{\left(e_u \cdot e_u\right)\left(e_v \cdot e_v\right) - \left(e_u \cdot e_v\right)^2} - \frac{\left(e_u \cdot e_v\right)^2 A_{vv}}{\left(e_u \cdot e_u\right)\left(e_v \cdot e_v\right) - \left(e_u \cdot e_v\right)^2}$$

$$A\left(e_u \wedge e_v\right) = \left(A_{uu}A_{vv} - A_{uv}A_{vu}\right)\boldsymbol{e_u} \wedge \boldsymbol{e_v}$$

$$A\left(e_u\right) \wedge A\left(e_v\right) = \left(A_{uu}A_{vv} - A_{uv}A_{vu}\right)\boldsymbol{e_u} \wedge \boldsymbol{e_v}$$

$$B = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_u}\right) = & B_{uu}\boldsymbol{e_u} + B_{vu}\boldsymbol{e_v} \\ L\left(\boldsymbol{e_v}\right) = & B_{uv}\boldsymbol{e_u} + B_{vv}\boldsymbol{e_v} \end{array} \right\}$$

$$A + B = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_u}\right) = & \left(A_{uu} + B_{uu}\right)\boldsymbol{e_u} + \left(A_{vu} + B_{vu}\right)\boldsymbol{e_v} \\ L\left(\boldsymbol{e_v}\right) = & \left(A_{uv} + B_{uv}\right)\boldsymbol{e_u} + \left(A_{vv} + B_{vv}\right)\boldsymbol{e_v} \end{array} \right\}$$

$$AB = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_u}\right) = & \left(A_{uu}B_{uu} + A_{uv}B_{vu}\right)\boldsymbol{e_u} + \left(A_{vu}B_{uu} + A_{vv}B_{vu}\right)\boldsymbol{e_v} \\ L\left(\boldsymbol{e_v}\right) = & \left(A_{uu}B_{uv} + A_{uv}B_{vv}\right)\boldsymbol{e_u} + \left(A_{vu}B_{uv} + A_{vv}B_{vv}\right)\boldsymbol{e_v} \end{array} \right\}$$

$$A - B = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_u}\right) = & \left(A_{uu} - B_{uu}\right)\boldsymbol{e_u} + \left(A_{vu} - B_{vu}\right)\boldsymbol{e_v} \\ L\left(\boldsymbol{e_v}\right) = & \left(A_{uv} - B_{uv}\right)\boldsymbol{e_u} + \left(A_{vv} - B_{vv}\right)\boldsymbol{e_v} \end{array} \right\}$$

$$a \cdot \overline{A}\left(b\right) - b \cdot \underline{A}\left(a\right) = 0$$

4d Minkowski spaqce (Space Time)

$$g = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\underline{T} = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_t}\right) = & T_{tt}\boldsymbol{e_t} + T_{xt}\boldsymbol{e_x} + T_{yt}\boldsymbol{e_y} + T_{zt}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_x}\right) = & T_{tx}\boldsymbol{e_t} + T_{xx}\boldsymbol{e_x} + T_{yx}\boldsymbol{e_y} + T_{zx}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_y}\right) = & T_{ty}\boldsymbol{e_t} + T_{xy}\boldsymbol{e_x} + T_{yy}\boldsymbol{e_y} + T_{zy}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_z}\right) = & T_{tz}\boldsymbol{e_t} + T_{xz}\boldsymbol{e_x} + T_{yz}\boldsymbol{e_y} + T_{zz}\boldsymbol{e_z} \end{array} \right\}$$

$$\overline{T} = \left\{ \begin{array}{ll} L\left(\boldsymbol{e_t}\right) = & T_{tt}\boldsymbol{e_t} - T_{tx}\boldsymbol{e_x} - T_{ty}\boldsymbol{e_y} - T_{tz}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_x}\right) = & -T_{xt}\boldsymbol{e_t} + T_{xx}\boldsymbol{e_x} + T_{xy}\boldsymbol{e_y} + T_{xz}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_y}\right) = & -T_{yt}\boldsymbol{e_t} + T_{yx}\boldsymbol{e_x} + T_{yy}\boldsymbol{e_y} + T_{yz}\boldsymbol{e_z} \\ L\left(\boldsymbol{e_z}\right) = & -T_{zt}\boldsymbol{e_t} + T_{zx}\boldsymbol{e_x} + T_{zy}\boldsymbol{e_y} + T_{zz}\boldsymbol{e_z} \end{array} \right\}$$

$$\operatorname{tr}\left(\underline{T}\right) = T_{tt} + T_{xx} + T_{yy} + T_{zz}$$

$$a \cdot \overline{T}\left(b\right) - b \cdot \underline{T}\left(a\right) = 0$$

3.9 Differential Operators

For the mathematical treatment of linear multivector differential operators see section 2.10.3. The is a differential operator class `Dop`. However, one never needs to use it directly. The operators are constructed from linear combinations of multivector products of the operators `Ga.grad` and `Ga.rgrad` as shown in the following code for both orthogonal rectangular and spherical 3-d coordinate systems.

Listing 3.4: python/Dop.py

```
from sympy import symbols, sin
from printer import Format,xpdf
from ga import Ga

Format()
coords = (x,y,z) = symbols('x y z',real=True)
(o3d,ex,ey,ez) = Ga.build('e*x|y|z',g=[1,1,1],coords=coords)
X = x*ex+y*ey+z*ez
I = o3d.i
v = o3d.mv('v','vector')
f = o3d.mv('f','scalar',f=True)
A = o3d.mv('A','vector',f=True)
dd = v|o3d.grad
lap = o3d.grad*o3d.grad
print r'\bm{X} =',X
print r'\bm{v} =',v
print r'\bm{A} =', A
print r'%\bm{v}\cdot\nabla =', dd
print r'%\nabla^{2} =',lap
print r'%\bm{v}\cdot\nabla f =',dd*f
print r'%\nabla^{2} f =',lap*f
print r'%\nabla^{2} \bm{A} =',lap*A
print r'%\bar{\nabla}\cdot v =', o3d.rgrad|v
Xgrad = X|o3d.grad
rgradX = o3d.rgrad|X
print r'%\bm{X}\cdot \nabla =', Xgrad
print r'%\bar{\nabla}\cdot \bm{X} =', rgradX
com = Xgrad - rgradX
print r'%\bm{X}\cdot \nabla - \bar{\nabla}\cdot \bm{X} =', com
```

```
sph_coords = (r,th,phi) = symbols('r theta phi',real=True)
(sp3d,er,eth,ephi) = Ga.build('e',g=[1,r**2,r**2*sin(th)**2],coords=sph_coo
f = sp3d.mv('f','scalar',f=True)
lap = sp3d.grad*sp3d.grad
print r'%\nabla^{2} = \nabla\cdot\nabla =', lap
print r'%\lp\nabla^{2}\rp f =', lap*f
print r'%\nabla\cdot\lp\nabla f\rp =',sp3d.grad|(sp3d.grad*f)
xpdf(paper='landscape',crop=True)
```

The output of this code is.

$$\boldsymbol{X} = x\boldsymbol{e_x} + y\boldsymbol{e_y} + z\boldsymbol{e_z}$$

$$\boldsymbol{v} = v^x\boldsymbol{e_x} + v^y\boldsymbol{e_y} + v^z\boldsymbol{e_z}$$

$$\boldsymbol{A} = A^x\boldsymbol{e_x} + A^y\boldsymbol{e_y} + A^z\boldsymbol{e_z}$$

$$\boldsymbol{v}\cdot\nabla = v^x\frac{\partial}{\partial x} + v^y\frac{\partial}{\partial y} + v^z\frac{\partial}{\partial z}$$

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

$$\boldsymbol{v}\cdot\nabla f = v^x\partial_x f + v^y\partial_y f + v^z\partial_z f$$

$$\nabla^2 f = \partial_x^2 f + \partial_y^2 f + \partial_z^2 f$$

$$\nabla^2\boldsymbol{A} = \left(\partial_x^2 A^x + \partial_y^2 A^x + \partial_z^2 A^x\right)\boldsymbol{e_x} + \left(\partial_x^2 A^y + \partial_y^2 A^y + \partial_z^2 A^y\right)\boldsymbol{e_y} + \left(\partial_x^2 A^z + \partial_y^2 A^z + \partial_z^2 A^z\right)\boldsymbol{e_z}$$

$$\bar{\nabla}\cdot v = v^x\frac{\partial}{\partial x} + v^y\frac{\partial}{\partial y} + v^z\frac{\partial}{\partial z}$$

$$\boldsymbol{X}\cdot\nabla = x\frac{\partial}{\partial x} + y\frac{\partial}{\partial y} + z\frac{\partial}{\partial z}$$

$$\bar{\nabla}\cdot\boldsymbol{X} = 3 + x\frac{\partial}{\partial x} + y\frac{\partial}{\partial y} + z\frac{\partial}{\partial z}$$

$$\boldsymbol{X}\cdot\nabla - \bar{\nabla}\cdot\boldsymbol{X} = -3$$

$$\nabla^2 = \nabla\cdot\nabla = \frac{2}{r}\frac{\partial}{\partial r} + \frac{\partial^2}{\partial r^2} + \frac{1}{r^2\tan(\theta)}\frac{\partial}{\partial\theta} + r^{-2}\frac{\partial^2}{\partial\theta^2} + \frac{1}{r^2\sin^2(\theta)}\frac{\partial^2}{\partial\phi^2}$$

$$\left(\nabla^2\right)f = \frac{1}{r^2}\left(r^2\partial_r^2 f + 2r\partial_r f + \partial_\theta^2 f + \frac{\partial_\theta f}{\tan(\theta)} + \frac{\partial_\phi^2 f}{\sin^2(\theta)}\right)$$

$$\nabla\cdot(\nabla f) = \frac{1}{r^2}\left(r^2\partial_r^2 f + 2r\partial_r f + \partial_\theta^2 f + \frac{\partial_\theta f}{\tan(\theta)} + \frac{\partial_\phi^2 f}{\sin^2(\theta)}\right)$$

Note that for print an operator in the IPython notebook one must implement (yet to be done) a printing method similar to `mv.Fmt()`.

3.10 Instantiating a Multi-linear Functions (Tensors)

The mathematical background for multi-linear functions is in section 2.12. To instantiate a multi-linear function use

`Mlt(self, f, Ga, nargs=None, fct=False)`

Where the arguments are

`f`	Either a string for a general tensor (this option is included mainly for debugging of the `Mlt` class) or a multi-linear function of manifold tangent vectors (multi-vectors of grade one) to scalar. For example one could generate a custom python function such as shown in `TensorDef.py`.
`Ga`	Geometric algebra that tensor is associated with.
`nargs`	If `f` is a string then `nargs` is the number of vector arguments of the tensor. If `f` is anything other than a string `nargs` is not required since `Mlt` determines the number of vector arguments from `f`.
`fct`	if `f` is a string then `fct=True` forces the tensor to be a tensor field (function of the coordinates. If `f` anything other than a string `fct` is not required since `Mlt` determines whether the tensor is a tensor field from `f`.

Listing 3.5: python/TensorDef.py

```
1
2 import sys
3 from sympy import symbols,sin,cos
4 from printer import Format,xpdf,Get_Program,Print_Function
5 from ga import Ga
6 from lt import Mlt
7
8 coords = symbols('t x y z',real=True)
9 (st4d,g0,g1,g2,g3) = Ga.build('gamma*t|x|y|z',g=[1,-1,-1,-1],coords=coords)
10
11 A = st4d.mv('T','bivector')
12
```

```
13 def TA(a1,a2):
14     global A
15     return A | (a1 ^ a2)
16
17 T = Mlt(TA,st4d) # Define multi-linear function
```

3.11 Basic Multilinear Function Class Functions

If we can instantiate multilinear functions we can use all the multilinear function class functions as described as follows. See section 2.12 for the mathematical description of each operation.

`self(kargs)`

Calling function to evaluates multilinear function for `kargs` list of vector arguments and returns a value. Note that a sympy scalar is returned, *not* a multilinear function.

`self.contract(slot1,slot2)`

Returns contraction of tensor between `slot1` and `slot2` where `slot1` is the index of the first vector argument and `slot2` is the index of the second vector argument of the tensor. For example if we have a rank two tensor, `T(a1,a2)`, then `T.contract(1,2)` is the contraction of `T`. For this case since there are only two slots there can only be one contraction.

`self.pdiff(slot)`

Returns gradient of tensor, `T`, with respect to slot vector. For example if the tensor is $T(a_1, a_2)$ then `T.pdiff(2)` is $\nabla_{a_2} T$. Since `T` is a scalar function, `T.pdiff(2)` is a vector function.

`self.cderiv()`

Returns covariant derivative of tensor field. If `T` is a tensor of rank k then `T.cderiv()` is a tensor of rank $k+1$. The operation performed is defined in section 2.12.

3.12 Standard Printing

Printing of multivectors is handled by the module `printer` which contains a string printer class derived from the *sympy* string printer class and a latex printer class derived from the *sympy* latex printer class. Additionally, there is an `Eprint` class that enhances the console output of *sympy* to make the printed output multivectors, functions, and derivatives more readable. `Eprint` requires an ansi console such as is supplied in linux or the program *ConEmu* replaces `cmd.exe`.

For a windows user the simplest way to implement *ConEmu* is to use the *geany* editor and in the Edit→Preferences→Tools menu replace `cmd.exe` with[8]

```
"C:\Program Files\ConEmu\ConEmu64.exe" /WndW 180 /cmd %c
```

and then run an example *galgeba* program that used `Eprint`. The default background and foreground colors make the output unreadable. To change these parameters to reasonable values:[9]

1. Right click on title bar of console.
2. Open *setting* window.
3. Open *colors* window.
4. Set the following parameters to the indicated values:

   Text: #0

   Back: #7

   Popup: #0

   Back: #7

   ☑ **Extend foreground colors with background** #13

If `Eprint` is called in a program (linux) when multivectors are printed the basis blades or bases are printed in bold text, functions are printed in red, and derivative operators in green.

For formatting the multivector output there is the member function

[8]The 180 in the *ConEmu* command line is the width of the console you wish to display in characters. Change the number to suit you.

[9]I am not exactly sure what the different parameter setting do. I achieved the result I wished for by trial and error. I encourage the users to experiment and share their results.

`Fmt(self,fmt=1,title=None)`

> `Fmt` is used to control how the multivector (`Mv`) is printed with the argument `fmt`. If `fmt=1` the entire multivector is printed on one line. If `fmt=2` each grade of the multivector is printed on one line. If `fmt=3` each component (base) of the multivector is printed on one line. If a `title` is given then `title=multivector` is printed. If the usual print command is used the entire multivector is printed on one line.

For formatting the tensor (`Mlt`) output there is the member function

`Fmt(self,cnt=1,title=None)`

> `Fmt` is used to control how the tensor is printed with the argument `cnt`. If `cnt=1` the each tensor component is printed on one line. If `fmt=n` n tensor components are printed on one line. If a `title` is given then `title=tensor` is printed. If the usual print command is used one tensor component is printed on one line. If `cnt` is greater or equal to the number of tensor components then the entire tensor is printer on one line.

For `Mv`, `Lt`, `Mlt`, and `Dop, Fmt()` returns a formatted string when executing from a script (`.py` file). When executing in *Ipython notebook* (`.ipynb` file) the function returns the *object* itself. For example if you wish to print a formatted multivector, `A`, from a script use

```
print A.Fmt(2,'A')
```

for an *Ipython notebook* session use

```
A.Fmt(2,'A')
```

these methods work in the same way for LaTeX printing.

3.13 Latex Printing

For latex printing one uses one functions from the `ga` module and one function from the `printer` module. The functions are

`Format(Fmode=True,Dmode=True)`

> This function from the `ga` module turns on latex printing with the following options

Argument	Value	Result
`Fmode`	`True`	Print functions without argument list, f
	`False`	Print functions with standard *sympy* latex formatting, $f\left(x,y,z\right)$
`Dmode`	`True`	Print partial derivatives with condensed notation, $\partial_x f$
	`False`	Print partial derivatives with standard *sympy* latex formatting $\frac{\partial f}{\partial x}$

`xpdf(filename=None,debug=False,paper=(14,11),crop=False)`

This function from the `printer` module post-processes the output captured from print statements, writes the resulting latex strings to the file `filename`, processes the file with pdflatex, and displays the resulting pdf file. All latex files except the pdf file are deleted. If `debug = True` the file `filename` is printed to standard output for debugging purposes and `filename` (the tex file) is saved. If `filename` is not entered the default filename is the root name of the python program being executed with `.tex` appended. The `paper` option defines the size of the paper sheet for latex. The format for the `paper` is

`paper=(w,h)`	`w` is paper width in inches and `h` is paper height in inches
`paper='letter'`	paper is standard letter size 8.5 in $\times$ 11 in
`paper='landscape'`	paper is standard letter size but 11 in $\times$ 8.5 in

The default of `paper=(14,11)` was chosen so that long multivector expressions would not be truncated on the display.

If the `crop` input is `True` the linux `pdfcrop` program is used to crop the pdf output (if output is one page). This only works for linux installations (where `pdfcrop` is installed).

The `xpdf` function requires that latex and a pdf viewer be installed on the computer.

`xpdf` *is not required when printing latex in IPython notebook.*

As an example of using the latex printing options when the following code is executed

```
1  from printer import Format, xpdf
2  from ga import Ga
3  Format()
4  g3d = Ga('e*x|y|z')
5  A = g3d.mv('A','mv')
6  print r'\bm{A} =',A
```

```
7       print A.Fmt(2,r'\bm{A}')
8       print A.Fmt(3,r'\bm{A}')
9       xpdf()
```

The following is displayed

$$\boldsymbol{A} = A + A^{x}\boldsymbol{e_x} + A^{y}\boldsymbol{e_y} + A^{z}\boldsymbol{e_z} + A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y} + A^{xz}\boldsymbol{e_x}\wedge\boldsymbol{e_z} + A^{yz}\boldsymbol{e_y}\wedge\boldsymbol{e_z} + A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z}$$

$$\begin{aligned}\boldsymbol{A} =& A \\ &+ A^{x}\boldsymbol{e_x} + A^{y}\boldsymbol{e_y} + A^{z}\boldsymbol{e_z} \\ &+ A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y} + A^{xz}\boldsymbol{e_x}\wedge\boldsymbol{e_z} + A^{yz}\boldsymbol{e_y}\wedge\boldsymbol{e_z} \\ &+ A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z}\end{aligned}$$

$$\begin{aligned}\boldsymbol{A} =& A \\ &+ A^{x}\boldsymbol{e_x} \\ &+ A^{y}\boldsymbol{e_y} \\ &+ A^{z}\boldsymbol{e_z} \\ &+ A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y} \\ &+ A^{xz}\boldsymbol{e_x}\wedge\boldsymbol{e_z} \\ &+ A^{yz}\boldsymbol{e_y}\wedge\boldsymbol{e_z} \\ &+ A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z}\end{aligned}$$

For the cases of derivatives the code is

```
1       from printer import Format, xpdf
2       from ga import Ga
3
4       Format()
5       X = (x,y,z) = symbols('x y z')
6       o3d = Ga('e_x e_y e_z',g=[1,1,1],coords=X)
7
8       f = o3d.mv('f','scalar',f=True)
9       A = o3d.mv('A','vector',f=True)
10      B = o3d.mv('B','grade2',f=True)
11
12      print r'\bm{A} =',A
13      print r'\bm{B} =',B
14
```

```
15      print 'grad*f =',o3d.grad*f
16      print r'grad|\bm{A} =',o3d.grad|A
17      (o3d.grad*A).Fmt(2,r'grad*\bm{A}')
18
19      print r'-I*(grad^\bm{A}) =',-o3g.mv_I*(o3d.grad^A)
20      print (o3d.grad*B).Fmt(2,r'grad*\bm{B}')
21      print r'grad^\bm{B} =',o3d.grad^B
22      print r'grad|\bm{B} =',o3d.grad|B
23
24      xpdf()
```

and the latex displayed output is (f is a scalar function)

$$\boldsymbol{A} = A^x \boldsymbol{e_x} + A^y \boldsymbol{e_y} + A^z \boldsymbol{e_z}$$
$$\boldsymbol{B} = B^{xy} \boldsymbol{e_x} \wedge \boldsymbol{e_y} + B^{xz} \boldsymbol{e_x} \wedge \boldsymbol{e_z} + B^{yz} \boldsymbol{e_y} \wedge \boldsymbol{e_z}$$
$$\boldsymbol{\nabla} f = \partial_x f \boldsymbol{e_x} + \partial_y f \boldsymbol{e_y} + \partial_z f \boldsymbol{e_z}$$
$$\boldsymbol{\nabla} \cdot \boldsymbol{A} = \partial_x A^x + \partial_y A^y + \partial_z A^z$$

$$\begin{aligned}\boldsymbol{\nabla} \boldsymbol{A} =& \partial_x A^x + \partial_y A^y + \partial_z A^z \\ &+ \left(-\partial_y A^x + \partial_x A^y\right) \boldsymbol{e_x} \wedge \boldsymbol{e_y} + \left(-\partial_z A^x + \partial_x A^z\right) \boldsymbol{e_x} \wedge \boldsymbol{e_z} + \left(-\partial_z A^y + \partial_y A^z\right) \boldsymbol{e_y} \wedge \boldsymbol{e_z}\end{aligned}$$

$$-I(\boldsymbol{\nabla} \wedge \boldsymbol{A}) = \left(-\partial_z A^y + \partial_y A^z\right) \boldsymbol{e_x} + \left(\partial_z A^x - \partial_x A^z\right) \boldsymbol{e_y} + \left(-\partial_y A^x + \partial_x A^y\right) \boldsymbol{e_z}$$

$$\begin{aligned}\boldsymbol{\nabla} \boldsymbol{B} =& \left(-\partial_y B^{xy} - \partial_z B^{xz}\right) \boldsymbol{e_x} + \left(\partial_x B^{xy} - \partial_z B^{yz}\right) \boldsymbol{e_y} + \left(\partial_x B^{xz} + \partial_y B^{yz}\right) \boldsymbol{e_z} \\ &+ \left(\partial_z B^{xy} - \partial_y B^{xz} + \partial_x B^{yz}\right) \boldsymbol{e_x} \wedge \boldsymbol{e_y} \wedge \boldsymbol{e_z}\end{aligned}$$

$$\boldsymbol{\nabla} \wedge \boldsymbol{B} = \left(\partial_z B^{xy} - \partial_y B^{xz} + \partial_x B^{yz}\right) \boldsymbol{e_x} \wedge \boldsymbol{e_y} \wedge \boldsymbol{e_z}$$
$$\boldsymbol{\nabla} \cdot \boldsymbol{B} = \left(-\partial_y B^{xy} - \partial_z B^{xz}\right) \boldsymbol{e_x} + \left(\partial_x B^{xy} - \partial_z B^{yz}\right) \boldsymbol{e_y} + \left(\partial_x B^{xz} + \partial_y B^{yz}\right) \boldsymbol{e_z}$$

This example also demonstrates several other features of the latex printer. In the case that strings are input into the latex printer such as `r'grad*\bm{A}'`, `r'grad^\bm{A}'`, or `r'grad*\bm{A}'`. The text symbols `grad`, `^`, `|`, and `*` are mapped by the `xpdf()` post-processor as follows if the string contains an `=`.

original	replacement	displayed latex
`grad*A`	`\bm{\nabla}A`	$\boldsymbol{\nabla} A$
`A^B`	`A\wedge B`	$A \wedge B$
`A|B`	`A\cdot B`	$A \cdot B$
`A*B`	`AB`	AB
`A<B`	`A\lfloor B`	$A \lfloor B$
`A>B`	`A\rfloor B`	$A \rfloor B$

If the first character in the string to be printed is a `%` none of the above substitutions are made before the latex processor is applied. In general for the latex printer strings are assumed to be in a math environment (equation or align) unless the first character in the string is a `#`.[10]

> Except where noted the conventions for latex printing follow those of the latex printing module of *sympy*. This includes translating *sympy* variables with Greek name (such as `alpha`) to the equivalent Greek symbol (α) for the purpose of latex printing. Also a single underscore in the variable name (such as "`X_j`") indicates a subscript (X_j), and a double underscore (such as "`X__k`") a superscript (X^k). The only other change with regard to the *sympy* latex printer is that matrices are printed full size (equation displaystyle).

3.13.1 Printing Lists/Tuples of Multivectors/Differential Operators

Since the expressions for multivectors or differential operators can be very long printing lists or tuples of such items can easily exceed the page with when printing in LaTeX or in "ipython notebook." I order to alleviate this problem the function `Fmt` can be used.

`Fmt(obj,fmt=0)`

This function from the `printer` module allows the formatted printing of lists/tuples or multivectors/differential operators.

`obj`	`obj` is a list or tuple of multivectors and/or differential operators.
`fmt=0`	`fmt=0` prints each element of the list/tuple on an individual lines[11].
	`fmt=1` prints all elements of the list/tuple on a single line[??].

If l is a list or tuple to print in the LaTeX environment use the command

[10] Preprocessing do not occur for the Ipython notebook and the string post processing commands `%` and `#` are not used in this case.

```
print Fmt(l) # One element of l per line
```

or

```
print Fmt(l,1) # All elements of l on one line
```

If you are printing in "ipython notebook" then enter

```
Fmt(l) # One element of l per line
```

or

```
Fmt(l,1) # All elements of l on one line
```

Bibliography

[1] Chris Doran and Anthony Lasenby, "Geometric Algebra for Physicists," Cambridge University Press, 2003. http://www.mrao.cam.ac.uk/~clifford

[2] David Hestenes and Garret Sobczyk, "Clifford Algebra to Geometric Calculus," Kluwer Academic Publishers, 1984. http://modelingnts.la.asu.edu

[3] Alan Macdonald, "Linear and Geometric Algebra," 2010. http://faculty.luther.edu/~macdonal/laga

[4] Alan Macdonald, "Vector and Geometric Calculus," 2012. http://faculty.luther.edu/~macdonal/vagc