Geometric Algebra Module for Sympy

Author: Alan Bromborsky

Abstract

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. In addition the module includes the geometric, outer (curl) and inner (div) derivatives and the ability to define a curvilinear coordinate system. The module requires the sympy module and the numpy module for numerical linear algebra calculations. For latex output a latex distribution must be installed.

What is Geometric Algebra?

Geometric algebra is the Clifford algebra of a real finite dimensional vector space or the algebra that results when a real finite dimensional 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 [Hestenes], [Doran]. 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 ([Doran],p85):

$$\begin{aligned} a(bc) &= (ab)c \\ a(b+c) &= ab+ac \\ (a+b)c &= ac+bc \\ aa &= a^2 \in \Re \end{aligned}$$

By induction the first three axioms also apply to any multivectors. The dot product of two vectors is defined by ([Doran],p86)

$$a \cdot b \equiv (ab+ba)/2$$

Then consider

$$\begin{aligned} c &= a+b \\ c^2 &= (a+b)^2 \\ c^2 &= a^2+ab+ba+b^2 \\ a\cdot b &= (c^2-a^2-b^2)/2 \in \Re \end{aligned}$$

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 $e_1, \ldots, e_n$ a complete basis for the geometric algebra is given by the scalar 1, the vectors $e_1, \ldots, e_n$ and all geometric products of vectors

$$e_{i_1}e_{i_2}\dots e_{i_r} \text{ where } 0 \le r \le n,\, 0 \le i_j \le n \text{ and } i_1 < i_2 < \dots < i_r$$

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

$$\boldsymbol{A} = A + \sum_{r=1}^{n} \sum_{i_1,\dots,i_r,\ \forall\ 0\le i_j\le n} A^{i_1,\dots,i_r} e_{i_1}e_{i_2}\dots e_r$$

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)

$$e_{i_{j+1}}e_{i_j} = 2e_{i_{j+1}} \cdot e_{i_j} - e_{i_j}e_{i_{j+1}} \tag{1}$$

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

$$\begin{aligned} e_3e_2e_1 &= (2(e_2\cdot e_3) - e_2e_3)e_1 \\ &= 2(e_2\cdot e_3)e_1 - e_2e_3e_1 \\ &= 2(e_2\cdot e_3)e_1 - e_2(2(e_1\cdot e_3) - e_1e_3) \\ &= 2((e_2\cdot e_3)e_1 - (e_1\cdot e_3)e_2) + e_2e_1e_3 \\ &= 2((e_2\cdot e_3)e_1 - (e_1\cdot e_3)e_2 + (e_1\cdot e_2)e_3) - e_1e_2e_3 \end{aligned}$$

which results from repeated application of equation (1). If the product of basis vectors contains repeated factors equation (1) can be used to bring the repeated factors next to one another so that if $e_{i_j} = e_{i_{j+1}}$ then $e_{i_j}e_{i_{j+1}} = e_{i_j} \cdot e_{i_{j+1}}$ 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 noncommutative 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 (binominal coefficient).

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

$$a_1 \wedge \dots \wedge a_r \equiv \sum_{i_{j_1}\dots i_{j_r}} \epsilon^{i_{j_1}\dots i_{j_r}} a_{i_{j_1}} \dots a_{i_{j_1}}$$

where $\epsilon^{i_{j_1}\dots i_{j_r}}$ 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 $a_1 \wedge \dots \wedge a_r$ 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

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

Using equation (2) 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.

$$\boldsymbol{B} = B + \sum_{r=1}^{n} \sum_{i_1,\dots,i_r,\ \forall\ 0\le i_j\le n} B^{i_1,\dots,i_r} e_{i_1}\wedge e_{i_2}\wedge\dots\wedge e_r$$

Using the blades $e_{i_1}\wedge e_{i_2}\wedge\dots\wedge e_r$ 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

$$\langle \boldsymbol{B}\rangle_r = \sum_{i_1,\dots,i_r,\ \forall\ 0\le i_j\le n} B^{i_1,\dots,i_r} e_{i_1}\wedge e_{i_2}\wedge\dots\wedge e_r$$

and

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

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

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

and define ([Hestenes],p6)

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

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

$$\boldsymbol{A}\wedge\boldsymbol{B} = \sum_{r,s} \langle \boldsymbol{A}\rangle_r \wedge \langle \boldsymbol{B}\rangle_s$$
$$\boldsymbol{A}\cdot\boldsymbol{B} = \sum_{r,s\ne 0} \langle \boldsymbol{A}\rangle_r \cdot \langle \boldsymbol{B}\rangle_s$$

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

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

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

$$\boldsymbol{A}^{\dagger} \equiv a_r\dots a_1$$

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

$$\boldsymbol{A}^{\dagger} = A + \sum_{r=1}^{n} (-1)^{r(r-1)/2} \sum_{i_1,\dots,i_r,\ \forall\ 0\le i_j\le n} A^{i_1,\dots,i_r} e_{i_1} \wedge e_{i_2} \wedge \cdots \wedge e_r$$

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

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

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 noncommutative symbols. We start by defining n noncommutative sympy symbols

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

Several software packages for numerical geometric algebra calculations are available from Doran-Lasenby group and the Dorst group. Symbolic packages for Clifford algebra using orthongonal bases such as $e_ie_j + e_je_i = 2\eta_{ij}$, 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, $e_1, e_2, \dots, e_n$ and a symbolic metric tensor $g_{ij} = e_i \cdot e_j$.

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 noncommuting sympy symbols.

The basic geometic algebra operations will be implemented in python by defining a multivector class, MV, and overloading the python operators in Table *1* where *A* and *B* are any two multivectors (In the case of +, -, *, ^, |, <<, and >> the operation is also defined if *A* or *B* is a sympy symbol or a sympy real number).

Operation	**Result**
A+B	sum of multivectors
A-B	difference of multivectors
*A*B*	geometric product
A^B	outer product of multivectors
A|B	inner product of multivectors
A<B	left contraction of multivectors
A>B	right contraction of multivectors

Table *1*. Multivector operations for GA

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

Note: 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).

Warning: 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 *define_precedence()* and *GAeval()* are available in the module *GAprecedence*.

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

Define the precedence of the multivector operations. The function *define_precedence()* 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

```
define_precedence(globals())
V = GAeval('X|Y^Z*W')
```

The sympy variable *V* would evaluate to *((X|Y)^Z)*W*.

Vector Basis and Metric

The two structures that define the `MV` (multivector) 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 multivector setup function. 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

$$g = \begin{bmatrix} (a0.a0) & (a0.a1) & (a0.a2) \\ (a0.a1) & (a1.a1) & (a1.a2) \\ (a0.a2) & (a1.a2) & (a2.a2) \end{bmatrix} \qquad (3)$$

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 equation (3) 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 static member of the multivector class `MV` as the sympy matrix *MV.metric* (g_{ij} = *MV.metric[i,j]*).

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

```
metric = '# # # 0 0, # # # 0 0, # # # 0 0, 0 0 0 0 2, 0 0 0 2 0'
```

then calling *MV.setup(basis,metric)* would initialize the metric tensor

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

Here we have specified that *n* and *nbar* are orthonal 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 *MV.setup* is called multivector representations of the basis local to the program are instantiated. For our first example that means that the symbolic vectors named *a0*, *a1*, and *a2* are created and returned from *MV.setup* via a tuple as in -

```
(a_1,a_2,a3) = MV.setup('a_1 a_2 a_3',metric=metric)
```

Note that the python variable name for a basis vector does not have to correspond to the name give in *MV.setup()*, one may wish to use a shorted python variable name to reduce programming (typing) errors, for example one could use -

```
(a1,a2,a3) = MV.setup('a_1 a_2 a_3',metric=metric)
```

or

```
(g1,g2,g3) = MV.setup('gamma_1 gamma_2 gamma_3',metric=metric)
```

so that if the latex printer is used *e1* would print as $\boldsymbol{e_1}$ and *g1* as $\boldsymbol{\gamma_1}$.

Note: Additionally *MV.setup* has simpified options for naming a set of basis vectors and for inputing an othogonal 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 othogonal basis if the signature of the vector space is $(1,1,1)$ (Euclidian 3-space) set *metric='[1,1,1]'* or if it is $(1,-1,-1,-1)$ (Minkowsi 4-space) set *metric='[1,-1,-1,-1]'*.

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 $a_{i_1}a_{i_2}\ldots a_{i_r}$ where $i_1 < i_2 < \cdots < i_r$ and $r \le n$. We call these multivectors bases and represent them internally with noncommutative symbols so for example $a_1a_2a_3$ is represented by

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

In the simplist case of two basis vectors *a_1* and *a_2* we have a list of bases

```
MV.bases = [[Symbol('ONE',commutative=False)],[Symbol('a_1',commutative=False),\
             Symbol('a_2',commutative=False)],[Symbol('a_1*a_2',commutative=False)]]
```

Note: The reason that the base for the scalar component of the multivector is defined as *Symbol('ONE',commutative=False)*, a noncommutative symbol is because of the properties of the left and right contraction operators which are non commutative if one is contracting a multivector with a scalar.

For the case of the basis blades we have

```
MV.blades = [[Symbol('ONE',commutative=False)],[Symbol('a_1',commutative=False),\
             Symbol('a_2',commutative=False)],[Symbol('a_1^a_2',commutative=False)]]
```

Note: 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 function that builds all the required arrays and dictionaries upto the base multiplication table is shown below. *MV.dim* is the number of basis vectors and the *combinations* functions from *itertools* constructs the index tupels for the bases of each pseudo grade. Then the noncommutative symbol representing each base is constructed from each index tuple. *MV.ONE* is the noncommutative symbol for the scalar base. For example if *MV.dim = 3* then

```
MV.index = ((),((0,),(1,),(2,)),((0,1),(0,2),(1,2)),((0,1,2)))
```

Note: 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.

```
@staticmethod
def build_base_blade_arrays(debug):
    indexes = tuple(range(MV.dim))
```

```
MV.index = [()]
for i in indexes:
    MV.index.append(tuple(combinations(indexes,i+1)))
MV.index = tuple(MV.index)

#Set up base and blade and index arrays

if not MV.is_orthogonal:
    MV.bases_flat = []
    MV.bases  = [MV.ONE]
    MV.base_to_index  = {MV.ONE:()}
    MV.index_to_base  = {():MV.ONE}
    MV.base_grades    = {MV.ONE:0}
    MV.base_grades[ONE] = 0

MV.blades = [MV.ONE]
MV.blades_flat = []
MV.blade_grades    = {MV.ONE:0}
MV.blade_grades[ONE] = 0
MV.blade_to_index = {MV.ONE:()}
MV.index_to_blade = {():MV.ONE}

ig = 1 #pseudo grade and grade index
for igrade in MV.index[1:]:
    if not MV.is_orthogonal:
        bases     = [] #base symbol array within pseudo grade
    blades    = [] #blade symbol array within grade
    ib = 0 #base index within grade
    for ibase in igrade:
        #build base name string
        (base_sym,base_str,blade_sym,blade_str) = MV.make_base_blade_symbol(ibase)

        if not MV.is_orthogonal:
            bases.append(base_sym)
            MV.bases_flat.append(base_sym)

        blades.append(blade_sym)
        MV.blades_flat.append(blade_sym)
        base_index = MV.index[ig][ib]

        #Add to dictionarys relating symbols and indexes
        if not MV.is_orthogonal:
            MV.base_to_index[base_sym]   = base_index
            MV.index_to_base[base_index] = base_sym
            MV.base_grades[base_sym]     = ig

        MV.blade_to_index[blade_sym] = base_index
        MV.index_to_blade[base_index] = blade_sym
        MV.blade_grades[blade_sym] = ig

        ib += 1
    ig += 1

    if not MV.is_orthogonal:
        MV.bases.append(tuple(bases))

    MV.blades.append(tuple(blades))

if not MV.is_orthogonal:
    MV.bases       = tuple(MV.bases)
    MV.bases_flat  = tuple(MV.bases_flat)
    MV.bases_flat1 = (MV.ONE,)+MV.bases_flat
    MV.bases_set   = set(MV.bases_flat[MV.dim:])

MV.blades       = tuple(MV.blades)
MV.blades_flat  = tuple(MV.blades_flat)
```

```
    MV.blades_flat1 = (MV.ONE,)+MV.blades_flat
    MV.blades_set   = set(MV.blades_flat[MV.dim:])

    return
```

Base Representation of Multivectors

In terms of the bases defined as noncommutative 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*MV.bases[0][0]+A__1*MV.bases[1][0]+A__2*MV.bases[1][1]+A__12*MV.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.

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

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

where A_r is a multivector of grade r and b is a vector. For our example basis the result is shown in Table *3*.

```
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 *3*. Bases blades in terms of bases.

The important thing to notice about Table *3* 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 *4*.

```
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 *4*. Bases in terms of basis blades.

Using Table *4* and simple substitution we can convert from a base multivector representation to a blade representation. Likewise, using Table *3* 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.bladeflg*, 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.

Warning: 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*.

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

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

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

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

where $\left\langle \right\rangle_t$ 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

$$A_r \cdot B_s = \left\langle A_r B_s \right\rangle_{|r-s|}$$

$$A_r \wedge B_s = \left\langle A_r B_s \right\rangle_{r+s}$$

and

$$A \cdot B = \sum_{r,s} A_r \cdot B_s$$

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

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

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

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

and

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

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

Warning: 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.

Warning: 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.

Reverse of Multivector

If A is the geometric product of r vectors

$$A = a_1\dots a_r$$

then the reverse of A designated $A^{\dagger}$ is defined by

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

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

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

but

$$\langle A\rangle_i^{\dagger} = (-1)^{\frac{i\left(i-1\right)}{2}}\langle A\rangle_i \tag{4}$$

which is proved by expanding the blade bases in terms of orthogonal vectors and showing that equation (4) 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 $\left(u^2 = \pm 1\right)$ that defines the plane of rotation.

Reciprocal Frames

If we have M linearly independent vectors (a frame), $a_1, \dots, a_M$, then the reciprocal frame is $a^1, \dots, a^M$ 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:

$$E_M = a_1 \wedge \dots \wedge a_M$$

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

Then

$$a^i = (-1)^{i-1}(a_1 \wedge \dots \wedge \breve{a}_i \wedge \dots \wedge a_M)E_M^{-1}$$

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 multivector setup function *MV.setup(basis,metric,rframe)* has the argument *rframe* with a default value of *False*. If it is set to *True* the reciprocal frame of the basis vectors is calculated. Additionally there is the function *reciprocal_frame(vlst,names='')* external to the *MV* class that will calculate the reciprocal frame of a list, *vlst*, of vectors. If the argument *names* is set to a space delimited string of names for the vectors the reciprocal vectors will be given these names.

Geometric Derivative

If F is a multivector field that is a function of a vector $x = x^i \boldsymbol{e}_i$ (we are using the summation convention that pairs of subscripts and superscripts are summed over the dimension of the vector space) then the geometric derivative ∇F is given by (in this section the summation convention is used):

$$\nabla F = \boldsymbol{e}^i \frac{\partial F}{\partial x^i}$$

If F_R is a grade-R multivector and $F_R = F_R^{i_1 \dots i_R} \boldsymbol{e}_{i_1} \wedge \dots \wedge \boldsymbol{e}_{i_R}$ then

$$\nabla F_R = \frac{\partial F_R^{i_1 \dots i_R}}{\partial x^j} \boldsymbol{e}^j \left(\boldsymbol{e}_{i_1} \wedge \dots \wedge \boldsymbol{e}_{i_R}\right)$$

Note that $\boldsymbol{e}^j \left(\boldsymbol{e}_{i_1} \wedge \dots \wedge \boldsymbol{e}_{i_R}\right)$ 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 defined as

$$\nabla \cdot F_R = \left\langle \nabla F_R \right\rangle_{R-1}$$

and

$$\nabla \wedge F_R = \left\langle \nabla F_R \right\rangle_{R+1}$$

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

Curvilinear coordinates are derived from a vector function $x(\boldsymbol{\theta})$ where $\boldsymbol{\theta} = (\theta_1, \dots, \theta_N)$ where the number of coordinates is equal to the dimension of the vector space. In the case of 3-dimensional spherical coordinates $\boldsymbol{\theta} = (r, \theta, \phi)$ and the coordinate generating function $x(\boldsymbol{\theta})$ is

$$x = r\cos(\phi)\sin(\theta)\boldsymbol{e_x} + r\sin(\phi)\sin(\theta)\boldsymbol{e_y} + r\cos(\theta)\boldsymbol{e_z}$$

A coordinate frame is derived from x by $\boldsymbol{e}_i = \frac{\partial x}{\partial \theta^i}$. The following show the frame for spherical coordinates.

$$\boldsymbol{e}_r = \cos(\phi)\sin(\theta)\boldsymbol{e_x} + \sin(\phi)\sin(\theta)\boldsymbol{e_y} + \cos(\theta)\boldsymbol{e_z}$$

$$\boldsymbol{e}_\theta = \cos(\phi)\cos(\theta)\boldsymbol{e_x} + r\cos(\theta)\sin(\phi)\boldsymbol{e_y} - r\sin(\theta)\boldsymbol{e_z}$$

$$\boldsymbol{e}_\phi = -r\sin(\phi)\sin(\theta)\boldsymbol{e_x} + r\cos(\phi)\sin(\theta)\boldsymbol{e_y}$$

The coordinate frame generated in this manner is not necessarily normalized so define a normalized frame by

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

This works for all $\boldsymbol{e}_i^2 \neq 0$ since we have defined $|\boldsymbol{e}_i| = \sqrt{|\boldsymbol{e}_i^2|}$. For spherical coordinates the normalized frame vectors are

$$\hat{\boldsymbol{e}}_r = \cos(\phi)\sin(\theta)\boldsymbol{e_x} + \sin(\phi)\sin(\theta)\boldsymbol{e_y} + \cos(\theta)\boldsymbol{e_z}$$

$$\hat{\boldsymbol{e}}_\theta = \cos(\phi)\cos(\theta)\boldsymbol{e_x} + \cos(\theta)\sin(\phi)\boldsymbol{e_y} - \sin(\theta)\boldsymbol{e_z}$$

$$\hat{\boldsymbol{e}}_\phi = -\sin(\phi)\boldsymbol{e_x} + \cos(\phi)\boldsymbol{e_y}$$

The geometric derivative in curvilinear coordinates is given by

$$\begin{aligned}
\nabla F_R &= \boldsymbol{e}^i \frac{\partial}{\partial x^i}\left(F_R^{i_1\dots i_R}\hat{\boldsymbol{e}}_{i_1}\wedge\dots\wedge\hat{\boldsymbol{e}}_{i_R}\right)\\
&= \boldsymbol{e^j}\frac{\partial}{\partial\theta^j}\left(F_R^{i_1\dots i_R}\hat{\boldsymbol{e}}_{i_1}\wedge\dots\wedge\hat{\boldsymbol{e}}_{i_R}\right)\\
&= \left(\frac{\partial}{\partial\theta^j}F_R^{i_1\dots i_R}\right)\boldsymbol{e^j}\left(\hat{\boldsymbol{e}}_{i_1}\wedge\dots\wedge\hat{\boldsymbol{e}}_{i_R}\right) + F_R^{i_1\dots i_R}\boldsymbol{e^j}\frac{\partial}{\partial\theta^j}\left(\hat{\boldsymbol{e}}_{i_1}\wedge\dots\wedge\hat{\boldsymbol{e}}_{i_R}\right)\\
&= \left(\frac{\partial}{\partial\theta^j}F_R^{i_1\dots i_R}\right)\boldsymbol{e^j}\left(\hat{\boldsymbol{e}}_{i_1}\wedge\dots\wedge\hat{\boldsymbol{e}}_{i_R}\right) + F_R^{i_1\dots i_R}C\left\{\hat{\boldsymbol{e}}_{i_1}\wedge\dots\wedge\hat{\boldsymbol{e}}_{i_R}\right\}
\end{aligned}$$

where

$$C\{\boldsymbol{\hat{e}}_{i_1}\wedge\cdots\wedge\boldsymbol{\hat{e}}_{i_R}\} = \boldsymbol{e}^j\,\frac{\partial}{\partial\theta^j}\left(\boldsymbol{\hat{e}}_{i_1}\wedge\cdots\wedge\boldsymbol{\hat{e}}_{i_R}\right)$$

are the connection multivectors for the curvilinear coordinate system. For a spherical coordinate system they are

$$C\{\boldsymbol{\hat{e}}_r\} = \frac{2}{r}$$

$$C\{\boldsymbol{\hat{e}}_\theta\} = \frac{\cos(\theta)}{r\sin(\theta)} + \frac{1}{r}\,\boldsymbol{\hat{e}}_r\wedge\boldsymbol{\hat{e}}_\theta$$

$$C\{\boldsymbol{\hat{e}}_\phi\} = \frac{1}{r}\,\boldsymbol{\hat{e}}_{\boldsymbol{r}}\wedge\boldsymbol{\hat{e}}_\phi + \frac{\cos(\theta)}{r\sin(\theta)}\,\boldsymbol{\hat{e}}_\theta\wedge\boldsymbol{\hat{e}}_\phi$$

$$C\{\hat{e}_r\wedge\hat{e}_\theta\} = -\frac{\cos(\theta)}{r\sin(\theta)}\,\boldsymbol{\hat{e}}_r + \frac{1}{r}\,\boldsymbol{\hat{e}}_\theta$$

$$C\{\boldsymbol{\hat{e}}_r\wedge\boldsymbol{\hat{e}}_\phi\} = \frac{1}{r}\,\boldsymbol{\hat{e}}_\phi - \frac{\cos(\theta)}{r\sin(\theta)}\,\boldsymbol{\hat{e}}_r\wedge\boldsymbol{\hat{e}}_\theta\wedge\boldsymbol{\hat{e}}_\phi$$

$$C\{\boldsymbol{\hat{e}}_\theta\wedge\boldsymbol{\hat{e}}_\phi\} = \frac{2}{r}\,\boldsymbol{\hat{e}}_r\wedge\boldsymbol{\hat{e}}_\theta\wedge\boldsymbol{\hat{e}}_\phi$$

$$C\{\boldsymbol{\hat{e}}_r\wedge\boldsymbol{\hat{e}}_\theta\wedge\boldsymbol{\hat{e}}_\phi\} = 0$$

Installation

To install the geometric algebra module on windows or linux perform the following operations

1. If not already installed install python 2.6 or 2.7 from <http://python.org/> or on windows you may wish to install the pythonxy package which includes python, mayavi2, and other pythonic tools (http://code.google.com/p/pythonxy/).

    1. To install sympy on linux or windows
        1. Go to <https://github.com/>
        2. Search for “sympy”
        3. Go to the “sympy/sympy” repository
        4. Download the repository zip file
        5. Uzip the file (anywhere on your machine).
        6. Open a terminal (console) in the root directory of the file
        7. Execute the command “python setup.py install” (put a “sudo” in front for linux)
    2. The alternative install for windows is -
        1. Get the latest sympy binary from `<http://code.google.com/p/sympy/downloads/list>’_ for windows.
        2. Execute binary on your system to install sympy.

2. To install texlive in linux or windows

   1. Go to <http://www.tug.org/texlive/acquire-netinstall.html> and click on "install-tl.zip" o download
   2. Unzip "install-tl.zip" anywhere on your machine
   3. Open the file "readme.en.html" in the "readme-html.dir" directory. This file contains the information needed to install texlive.
   4. Open a terminal (console) in the "install-tl-XXXXXX" directory
   5. Follow the instructions in "readme.en.html" file to run the install-tl.bat file in windows or the install-tl script file in linux.

3. Install python-nympy (for windows go to <http://sourceforge.net/projects/numpy/files/NumPy/1.6.2/> and install the distribution of numpy appropriate for your system)

4. Install "GA.zip" anywhere on your computer and unzip.

5. It is strongly suggested that you go to <http://www.geany.org/Download/Releases> and install the version of the "geany" editor appropriate for your system.

6. If you wish to use "enhance_print" on windows -

   1. Go to <https://github.com/adoxa/ansicon/downloads> and download "ansicon"
   2. In the Edit -> Preferences -> Tools menu of "geany" enter into the Terminal input the full path of "ansicon.exe"

After installation if you are doing you code development in the GA directory you need only include

```
from GAPrint import xdvi,enhance_print
from GA import *
```

to use the GA module. If you are working in another directory the GA directory must be in the python path or you can use the `sys.path.append('path to GA')` command in your program to access GA or include a .pth file in the appropriate location (http://docs.python.org/library/site.html).

In addition to the code shown in the examples section of this document there are more examples in the Examples directory under the GA directory.

Module Components

Initializing Multivector Class

The multivector class is initialized with:

`MV.setup`(*basis*, *metric=None*, *coords=None*, *rframe=False*, *debug=False*, *curv=(None, None)*)

The *basis* and *metric* parameters were described in section *Vector Basis and Metric*. If *rframe=True* the reciprocal frame of the symbolic bases vectors is calculated. If *debug=True* the data structure required to initialize the `MV` class are printer out. *coords* is a tuple of `sympy` symbols equal in length to the number of basis vectors. These symbols are used as the arguments of a multivector field as a function of position and for calculating the derivatives of a multivector field (if *coords* is defined then *rframe* is automatically set equal to *True*). Additionally, `MV.setup()` calculates the pseudo scalar, I and makes them available to the programmer as *MV.I* and *MV.Iinv*.

`MV.setup()` always returns a tuple containing the basis vectors (as multivectors) so that if we have the code

```
(e1,e2,e3) = MV.setup('e_1 e_2 e_3')
```

then we can define a multivector by the expression

```
(a1,a2,a3) = symbols('a__1 a__2 a__3')
A = a1*e1+a2*e2+a3*e3
```

if *coords* is defined then `MV.setup()` returns the tuple

```
X = (x1,x2,x3) = symbols('x__1 x__2 x__3')
(e1,e2,e3,grad) = MV.setup('e_1 e_2 e_3',coords=X)
```

then the additional vector *grad* is returned. *grad* acts as the gradient operator (geometric derivative) so that if `F()` is a multivector function of *(x,y,z)* then

```
DFl = grad*F
DFr = F*grad
```

are the left and right geometric derivatives of `F()`.

The final parameter in `MV.setup()` is *curv* which defines a curvilinear coordinate system. If 3-dimensional spherical coordinates are required we would define -

```
X = (r,th,phi) = symbols('r theta phi')
curv = [[r*cos(phi)*sin(th),r*sin(phi)*sin(th),r*cos(th)],[1,r,r*sin(th)]]
(er,eth,ephi,grad) = MV.setup('e_r e_theta e_phi',metric='[1,1,1]',coords=X,curv=curv)
```

The first component of *curv* is

```
[r*cos(phi)*sin(th),r*sin(phi)*sin(th),r*cos(th)]
```

This is the position vector for the spherical coordinate system expressed in terms of the rectangular coordinate components given in terms of the spherical coordinates *r*, *th*, and *phi*. The second component of *curv* is

```
[1,r,r*sin(th)]
```

The components of *curv[1]* are the normalizing factors for the basis vectors of the spherical coordinate system that are calculated from the derivatives of *curv[0]* with respect to the coordinates *r*, *th*, and *phi*. In theory the normalizing factors can be calculated from the derivatives of *curv[0]*. In practice one cannot currently specify in sympy that the square of a function is always positive which leads to problems when the normalizing factor is the square root of a squared function. To avoid these problems the normalizing factors are explicitly defined in *curv[1]*.

Note: In the case of curvlinear coordinates *debug* also prints the connection multivectors.

Instantiating a Multivector

Now that grades and bases have been described we can show all the ways that a multivector can be instantiated. As an example assume that the multivector space is initialized with

```
(e1,e2,e3) = MV.setup('e_1 e_2 e_3')
```

then multivectors could be instantiated with

```
(a1,a2,a3) = symbols('a__1 a__2 a__3')
x = a1*e1+a2*e2+a3*e3
```

```
y = x*e1*e2
z = x|y
w = x^y
```

or with the multivector class constructor:

class `MV`(*base=None*, *mvtype=None*, *fct=False*, *blade_rep=True*)

base is a string that defines the name of the multivector for output purposes. *base* and *mvtype* are defined by the following table and *fct* is a switch that will convert the symbolic coefficients of a multivector to functions if coordinate variables have been defined when `MV.setup()` is called:

mvtype	**base**	**result**
default	default	Zero multivector
'scalar'	string s	symbolic scalar of value Symbol(s)
'vector	string s	symbolic vector
'grade2'	string s	symbolic bivector
'grade'	string s,n	symbolic n-grade multivector
'pseudo'	string s	symbolic pseudoscalar
'spinor'	string s	symbolic even multivector
'mv'	string s	symbolic general multivector
default	sympy scalar c	zero grade multivector with coefficient c
default	multivector	copy constructor for multivector

If the *base* argument is a string s then the coefficients of the resulting multivector are named as follows:

The grade r coefficients consist of the base string, s, followed by a double underscore, __, and an index string of r symbols. If *coords* is defined the index string will consist of coordinate names in a normal order defined by the *coords* tuple. If *coords* is not defined the index string will be integers in normal (ascending) order (for an n dimensional vector space the indices will be 1 to n). The double underscore is used because the latex printer interprets it as a superscript and superscripts in the coefficients will balance subscripts in the bases.

For example if If *coords=(x,y,z)* and the base is *A*, the list of all possible coefficients for the most general multivector would be *A*, *A__x*, *A__y*, *A__z*, *A__xy*, *A__xz*, *A__yz*, and *A_xyz*. If the latex printer is used and *e* is the base for the basis vectors then the pseudo scalar would print as $A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z}$. If coordinates are not defined it would print as $A^{123}\boldsymbol{e_1}\wedge\boldsymbol{e_2}\wedge\boldsymbol{e_3}$. For printed output all multivectors are represented in terms of products of the basis vectors, either as geometric products or wedge products. This is also true for the output of expressions containing reciprocal basis vectors.

If the *fct* argument of `MV()` is set to *True* and the *coords* argument in `MV.setup()` is defined the symbolic coefficients of the multivector are functions of the coordinates.

Basic Multivector Class Functions

`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.

`dd`(*self*, *v*)
: For a mutivector function *F* and a vector *v* then *F.dd(v)* is the directional derivate of *F* in the direction *v*, $(v\cdot\nabla)F$.

`diff`(*self*, *var*)
: Calculate derivative of each multivector coefficient with resepect to variable *var* and form new multivector from coefficients.

`dual`(*self*)
: Return dual of multivector which is multivector left multiplied by pseudoscalar *MV.I* ([Hestenes],p22).

`even`(*self*)
: Return the even grade components of the multivector.

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

`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 if *self*sefl.rev()* is a nonzero ctor.

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

`simplify`(*self*)
: Return multivector where sympy simplify function has been applied to each coefficient of the multivector.

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

`rev`(*self*)
: Return the reverse of the multivector. See section *Reverse of Multivector*.

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

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

`trigsimp`(*self*, ***kwargs*)

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

Basic Multivector Functions

`Com`(*A*, *B*)

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

`DD`(*v*, *f*)

Calculate directional derivative of multivector function *f* in direction of vector *v*. Returns *f.dd(v)*.

`Format`(*Fmode=True*, *Dmode=True*, *ipy=False*)

See latex printing.

`GAeval`(*s*, *pstr=False*)

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

`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.

`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 the coefficient to normalize the vectors.

`ScalarFunction`(*TheFunction*)

If *TheFuction* is a real *sympy* fuction a scalar multivector function is returned.

`cross`(*M1*, *M2*)

If *M1* and *M2* are 3-dimensional euclidian vectors the vector cross product is returned, $v_1 \times v_2 = -I(v_1 \wedge v_2)$.

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

This is used with the *GAeval()* fuction to evaluate a string representing a multivector expression with a revised operator precedence. *define_precedence()* redefines the operator precedence for multivectors. *define_precedence()* 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 ([Hestenes],p7,[Doran]_,p38).

`dual`(*M*)

Return the dual of the multivector *M*, MI^{-1}.

`inv`(*B*)

If for the multivector B, $BB^{\dagger}$ is a nonzero scalar, return $B^{-1} = B^{\dagger}/(BB^{\dagger})$.

`proj`(*B*, *A*)

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

`refl`(*B*, *A*)

Reflect blade *A* in blade *B*. If *r* is grade of *A* and *s* is grade of *B* returns $(-1)^{s(r+1)}BAB^{-1}$.

`rot`(*itheta*, *A*)

Rotate blade *A* by 2-blade *itheta*. Is is assumed that *itheta*itheta > 0* so that the rotation is Euclidian and not hyperbolic so that the angle of rotation is *theta = itheta.norm()*. Ther in 3-dimensional Euclidian space. *theta* is the angle of rotation (scalar in radians) and *n* is the vector axis of rotation. Returned is the rotor *cos(theta)+sin(theta)*N* where *N* is the normalized dual of *n*.

Multivector Derivatives

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

```
X = (x,y,z) = symbols('x y z')
(ex,ey,ez,grad) = MV.setup('e*x|y|z',metric='[1,1,1]',coords=X)
```

Then the gradient operator vector is *grad* (actually the user can give it any name he wants to). Then the derivatives of the multivector function *F* are given by

```
F = MV('F','mv',fct=True)
```

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

The preceding code block gives examples of all possible multivector derivatives of the multivector function *F* where * give the left and right geometric derivatives, ^ gives the left and right exterior (curl) derivatives, | gives the left and right interior (div) derivatives, < give the left and right derivatives for the left contraction, and > give the left and right derivatives for the right contraction. To understand the left and right derivatives see a reference on geometric calculus ([Doran],chapter6).

If one is taking the derivative of a complex expression that expression should be in parenthesis. Additionally, whether or not one is taking the derivative of a complex expression the *grad* vector and the expression it is operating on should always be in parenthesis unless the grad operator and the expression it is operating on are the only objects in the expression.

Vector Manifolds

In addtition to the *GA* module there is a *Manifold* module that allows for the definition of a geometric algebra and calculus on a vector manifold. The vector mainfold is defined by a vector function of some coordinates in an embedding vector space ([Doran],p202,[Hestenes]_,p139). For example the unit 2-sphere would be the collection of vectors on the unit shpere in 3-dimensions with possible coordinates of θ and ϕ the angles of elevation and azimuth. A vector function $X(\theta,\phi)$ that defines the manifold would be given by

$$X(\theta,\phi) = \cos(\theta)\boldsymbol{e_z} + \cos(\theta)\left(\cos(\phi)\boldsymbol{e_x} + \sin(\phi)\boldsymbol{e_y}\right)$$

The module *Manifold.py* is transitionary in that all calculation are performed in the embedding vector space (geometric algebra). Thus due to the limitations on *sympy*'s *simplify()* and *trigsimp()*, simple expressions may appear to be very complicated since they are expressed in terms of the basis vectors (bases/blades) of the embedding space and not in terms of the vector space (geometric algebra) formed from the manifold's basis vectors. A future implementation of *Manifold.py* will correct this difficiency. The member functions of the vector manifold follow.

`Manifold`(*x*, *coords*, *debug=False*, *I=None*)

Initializer for vector manifold where *x* is the vector function of the *coords* that defines the manifold and *coords* is the list/tuple of sympy symbols that are the coordinates. The basis vectors of the manifold as a fuction of the coordinates are returned as a tuple. *I* is the pseudo scalar for the manifold. The default is for the initializer to calculate *I*, however for complicated *x* functions (especially where trigonometric functions of the coordinates are involved) it is sometimes a good idea to calculate *I* separately and input it to *Manifold()*.

`Basis`(*self*)

Return the basis vectors of the manifold as a tuple.

`DD`(*self*, *v*, *F*, *opstr=False*)

Return the manifold directional derivative of a multivector function *F* defined on the manifold in the vector direction *v*.

`Grad`(*self*, *F*)

Return the manifold multivector derivative of the multivector function *F* defined on the manifold.

`Proj`(*self*, *F*)

Return the projection of the multivector *F* onto the manifold tangent space.

An example of a simple vector manifold is shown below which demonstrates the instanciation of a manifold, the defining of vector and scalar functions on the manifold and the calculation of the geometric derivative of those functions.

```
def Simple_manifold_with_vector_function_derivative():
    print_function()
    coords = (x,y,z) = symbols('x y z')
    basis = (ex, ey, ez, grad) = MV.setup('e_x e_y e_z',metric='[1,1,1]',coords=coords)
    # Define surface
    mfvar = (u,v) = symbols('u v')
    X = u*ex+v*ey+(u**2+v**2)*ez
    print '\\mbox{Manifold Definiton: } X =',X
    MF = Manifold(X,mfvar)
    (eu,ev) = MF.Basis()
    # Define field on the surface.
    g = (v+1)*log(u)
    print '\\mbox{Scalar Function: } g =',g
    dg = MF.grad*g
    print '\\mbox{Scalar Function Derivative: } \\nabla g =',dg
    print '\\eval{\\nabla g}{(1,0)} =',dg.subs({u:1,v:0})
    # Define vector field on the surface
    G = v**2*eu+u**2*ev
    print '\\mbox{Vector Function: } G =',G
    dG = MF.grad*G
    print '\\mbox{Vector Function Derivative: } \\nabla G =',dG
    print '\\eval{\\nabla G}{(1,0)} =',dG.subs({u:1,v:0})
    return
```

Code Output:

$$\mbox{Manifold Definiton: } X = u\boldsymbol{e}_{x} + v\boldsymbol{e}_{y} + \left(u^{2} + v^{2}\right)\boldsymbol{e}_{z}$$

$$\mbox{Scalar Function: } g = (v+1)\log(u)$$

$$\mbox{Scalar Function Derivative: } \nabla g = \left(\frac{-4u^{2}v\log(u) + 4v^{3} + 4v^{2} + v + 1}{u\left(4u^{2} + 4v^{2} + 1\right)}\right)\boldsymbol{e}_{x} + \left(\frac{4u^{2}\log(u) - 4v^{2} - 4v + \log(u)}{4u^{2} + 4v^{2} + 1}\right)\boldsymbol{e}_{y} + \left(2\frac{v\log(u) + v + 1}{4u^{2} + 4v^{2} + 1}\right)\boldsymbol{e}_{z}$$

$$\left.\nabla g\right|_{(1,0)} = \frac{1}{5}\boldsymbol{e}_{x} + \frac{2}{5}\boldsymbol{e}_{z}$$

$$\mbox{Vector Function: } G = v^{2}\boldsymbol{e}_{x} + u^{2}\boldsymbol{e}_{y} + \left(2uv\left(u+v\right)\right)\boldsymbol{e}_{z}$$

$$\mbox{Vector Function Derivative: } \nabla G = 4\frac{uv\left(u+v\right)}{4u^{2} + 4v^{2} + 1} + \left(2\frac{-4u^{2}v + 4uv^{2} + u - v}{4u^{2} + 4v^{2} + 1}\right)\boldsymbol{e}_{x}\wedge\boldsymbol{e}_{y} + \left(2\frac{v\left(-4u^{3} - 8u^{2}v + 8uv^{2} + 2u + 4v^{3} - v\right)}{4u^{2} + 4v^{2} + 1}\right)\boldsymbol{e}_{x}\wedge\boldsymbol{e}_{z} + \left(2\frac{u\left(4u^{3} + 8u^{2}v - 8uv^{2} - u - 4v^{3} + 2v\right)}{4u^{2} + 4v^{2} + 1}\right)\boldsymbol{e}_{y}\wedge\boldsymbol{e}_{z}$$

$$\left.\nabla G\right|_{(1,0)} = \frac{2}{5}\boldsymbol{e}_{x}\wedge\boldsymbol{e}_{y} + \frac{6}{5}\boldsymbol{e}_{y}\wedge\boldsymbol{e}_{z}$$

Standard Printing

Printing of multivectors is handled by the module *GAPrint* 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 *enhanced_print* class that enhances the console output of sympy to make the printed output multivectors, functions, and derivatives more readable. *enhanced_print* requires an ansi console such as is supplied in linux or the program *ansicon* (github.com/adoxa/ansicon) for windows which replaces *cmd.exe*.

For a windows user the simplest way to implement ansicon is to use the *geany* editor and in the Edit->Preferences->Tools menu replace *cmd.exe* with *ansicon.exe* (be sure to supply the path to *ansicon*).

If *enhanced_print* 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

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

Fmt is used to control how the multivector 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.

Latex Printing

For latex printing one uses one functions from the *GA* module and one function from the *GAPrint* module. The functions are

`Format`(*Fmode=True*, *Dmode=True*, *ipy=False*)

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, $f(x,y,z)$
Dmode	*True*	Print partial derivatives with condensed notatation, $\partial_x f$
	False	Print partial derivatives with standard sympy latex formatting $\frac{\partial f}{\partial x}$
ipy	*False*	Redirect print output to file for post-processing by latex
	True	Do not redirect print output. This is used for Ipython with MathJax

`xdvi`(*filename=None*, *pdf=''*, *debug=False*)

This function from the *GAPrint* module post-processes the output captured from print statements. Write the resulting latex strings to the file *filename*, processes the file with pdflatex, and displays the resulting pdf file. *pdf* is the name of the pdf viewer on your computer. If you are running *ubuntu* the *evince* viewer is automatically used. On other operating systems if *pdf = ''* the name of the pdf file is executed. If the pdf file type is associated with a viewer this will launch the viewer with the associated 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 **xdvi** function requires that latex and a pdf viewer be installed on the computer.

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

```
from GAPrint import xdvi
from GA import *
Format()
(ex,ey,ez) = MV.setup('e*x|y|z')
A = MV('A','mv')
print r'\bm{A} =',A
```

```
A.Fmt(2,r'\bm{A}')
A.Fmt(3,r'\bm{A}')

xdvi()
```

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

```
from GAPrint import xdvi
from GA import *

Format()
X = (x,y,z) = symbols('x y z')
(ex,ey,ez,grad) = MV.setup('e_x e_y e_z',metric='[1,1,1]',coords=X)

f = MV('f','scalar',fct=True)
A = MV('A','vector',fct=True)
B = MV('B','grade2',fct=True)

print r'\bm{A} =',A
print r'\bm{B} =',B

print 'grad*f =',grad*f
print r'grad|\bm{A} =',grad|A
print r'grad*\bm{A} =',grad*A

print r'-I*(grad^\bm{A}) =',-MV.I*(grad^A)
print r'grad*\bm{B} =',grad*B
print r'grad^\bm{B} =',grad^B
print r'grad|\bm{B} =',grad|B

xdvi()
```

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

$$
\begin{aligned}
\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} \\
\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}} \\
-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}} \\
\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}} \\
\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}}
\end{aligned}
$$

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 *xdvi()* 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 string to be printed contains 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 string contains a #.

Note: 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).

Other Printing Functions

These functions are used together if one wishes to print both code and output in a single file. They work for text printing and for latex printing.

For these functions to work properly the last function defined must not contain a *print_function()* call (the last function defined is usually a *dummy()* function that does nothing).

Additionally, to work properly none of the functions containing *print_function()* can contain function definintions (local functions).

`get_program`(*off=False*)

Tells program to print both code and output from functions that have been properly tagged with *print_function()*. *get_program()* must be in main program before the functions that you wish code printing from are executed. the *off* argument in *get_program()* allows one to turn off the printing of the code by changing one line in the entire program (*off=True*).

`print_function`()

print_function() is included in those functions where one wishes to print the code block along with (before) the usual printed output. The *print_function()* statement should be included immediately after the function def statement. For proper usage of both *print_function()* and *get_program()* see the following example.

As an example consider the following code

```
from GAPrint import xdvi,get_program,print_function
from GA import *

Format()

def basic_multivector_operations_3D():
    print_function()
    (ex,ey,ez) = MV.setup('e*x|y|z')

    A = MV('A','mv')

    A.Fmt(1,'A')
    A.Fmt(2,'A')
    A.Fmt(3,'A')

    A.even().Fmt(1,'%A_{+}')
    A.odd().Fmt(1,'%A_{-}')

    X = MV('X','vector')
    Y = MV('Y','vector')

    print 'g_{ij} =',MV.metric

    X.Fmt(1,'X')
    Y.Fmt(1,'Y')

    (X*Y).Fmt(2,'X*Y')
    (X^Y).Fmt(2,'X^Y')
    (X|Y).Fmt(2,'X|Y')
    return

def basic_multivector_operations_2D():
    print_function()
    (ex,ey) = MV.setup('e*x|y')

    print 'g_{ij} =',MV.metric

    X = MV('X','vector')
    A = MV('A','spinor')

    X.Fmt(1,'X')
    A.Fmt(1,'A')

    (X|A).Fmt(2,'X|A')
    (X<A).Fmt(2,'X<A')
    (A>X).Fmt(2,'A>X')
    return

def dummy():
    return

get_program()

basic_multivector_operations_3D()
basic_multivector_operations_2D()

xdvi()
```

The latex output of the code is

```
def basic_multivector_operations_3D():
    print_function()
    (ex,ey,ez) = MV.setup('e*x|y|z')
    A = MV('A','mv')
    A.Fmt(1,'A')
    A.Fmt(2,'A')
    A.Fmt(3,'A')
    A.even().Fmt(1,'%A_{+}')
    A.odd().Fmt(1,'%A_{-}')
    X = MV('X','vector')
    Y = MV('Y','vector')
    print 'g_{ij} =',MV.metric
    X.Fmt(1,'X')
    Y.Fmt(1,'Y')
    (X*Y).Fmt(2,'X*Y')
    (X^Y).Fmt(2,'X^Y')
    (X|Y).Fmt(2,'X|Y')
    return
```

Code Output:

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

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

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

$$A_{+} = A + A^{xy}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} + A^{xz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} + A^{yz}\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}}$$

$$A_{-} = A^{x}\boldsymbol{e_{x}} + A^{y}\boldsymbol{e_{y}} + A^{z}\boldsymbol{e_{z}} + A^{xyz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}}$$

$$g_{ij} = \left[\begin{smallmatrix} (e_{x}\cdot e_{x}) & (e_{x}\cdot e_{y}) & (e_{x}\cdot e_{z}) \\ (e_{x}\cdot e_{y}) & (e_{y}\cdot e_{y}) & (e_{y}\cdot e_{z}) \\ (e_{x}\cdot e_{z}) & (e_{y}\cdot e_{z}) & (e_{z}\cdot e_{z}) \end{smallmatrix}\right]$$

$$X = X^{x}\boldsymbol{e_{x}} + X^{y}\boldsymbol{e_{y}} + X^{z}\boldsymbol{e_{z}}$$

$$Y = Y^{x}\boldsymbol{e_{x}} + Y^{y}\boldsymbol{e_{y}} + Y^{z}\boldsymbol{e_{z}}$$

$$\begin{aligned} XY =&X^{x}Y^{x}\left(e_{x}\cdot e_{x}\right) + X^{x}Y^{y}\left(e_{x}\cdot e_{y}\right) + X^{x}Y^{z}\left(e_{x}\cdot e_{z}\right) + X^{y}Y^{x}\left(e_{x}\cdot e_{y}\right) + X^{y}Y^{y}\left(e_{y}\cdot e_{y}\right) + X^{y}Y^{z}\left(e_{y}\cdot e_{z}\right) + X^{z}Y^{x}\left(e_{x}\cdot e_{z}\right) + X^{z}Y^{y}\left(e_{y}\cdot e_{z}\right) + X^{z}Y^{z}\left(e_{z}\cdot e_{z}\right) \\ &+ \left(X^{x}Y^{y} - X^{y}Y^{x}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} + \left(X^{x}Y^{z} - X^{z}Y^{x}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} + \left(X^{y}Y^{z} - X^{z}Y^{y}\right)\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} \end{aligned}$$

$$X\wedge Y = \left(X^{x}Y^{y} - X^{y}Y^{x}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} + \left(X^{x}Y^{z} - X^{z}Y^{x}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} + \left(X^{y}Y^{z} - X^{z}Y^{y}\right)\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}}$$

```
def basic_multivector_operations_2D():
    print_function()
    (ex,ey) = MV.setup('e*x|y')
    print 'g_{ij} =',MV.metric
    X = MV('X','vector')
    A = MV('A','spinor')
    X.Fmt(1,'X')
    A.Fmt(1,'A')
    (X|A).Fmt(2,'X|A')
    (X<A).Fmt(2,'X<A')
    (A>X).Fmt(2,'A>X')
    return
```

Code Output:

$$g_{ij} = \left[\begin{smallmatrix}(e_x\cdot e_x) & (e_x\cdot e_y)\\ (e_x\cdot e_y) & (e_y\cdot e_y)\end{smallmatrix}\right]$$

$$X = X^{x}\boldsymbol{e_{x}} + X^{y}\boldsymbol{e_{y}}$$

$$A = A + A^{xy}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}}$$

$$X\cdot A = \left(A^{xy}\left(-X^{x}\left(e_{x}\cdot e_{y}\right) - X^{y}\left(e_{y}\cdot e_{y}\right)\right)\right)\boldsymbol{e_{x}} + \left(A^{xy}\left(X^{x}\left(e_{x}\cdot e_{x}\right) + X^{y}\left(e_{x}\cdot e_{y}\right)\right)\right)\boldsymbol{e_{y}}$$

$$X\rfloor A = \left(A^{xy}\left(-X^{x}\left(e_{x}\cdot e_{y}\right) - X^{y}\left(e_{y}\cdot e_{y}\right)\right)\right)\boldsymbol{e_{x}} + \left(A^{xy}\left(X^{x}\left(e_{x}\cdot e_{x}\right) + X^{y}\left(e_{x}\cdot e_{y}\right)\right)\right)\boldsymbol{e_{y}}$$

$$A\lfloor X = \left(A^{xy}\left(X^{x}\left(e_{x}\cdot e_{y}\right) + X^{y}\left(e_{y}\cdot e_{y}\right)\right)\right)\boldsymbol{e_{x}} + \left(A^{xy}\left(-X^{x}\left(e_{x}\cdot e_{x}\right) - X^{y}\left(e_{x}\cdot e_{y}\right)\right)\right)\boldsymbol{e_{y}}$$

Examples

Algebra

BAC-CAB Formulas

This example demonstrates the most general metric tensor

$$g_{ij} = \begin{bmatrix} (a\cdot a) & (a\cdot b) & (a\cdot c) & (a\cdot d)\\ (a\cdot b) & (b\cdot b) & (b\cdot c) & (b\cdot d)\\ (a\cdot c) & (b\cdot c) & (c\cdot c) & (c\cdot d)\\ (a\cdot d) & (b\cdot d) & (c\cdot d) & (d\cdot d) \end{bmatrix}$$

and how the *GA* module can be used to verify and expand geometric algebra identities consisting of relations between the abstract vectors a, b, c, and d.

```
from GAPrint import xdvi
from GA import *
Format()

(a,b,c,d) = MV.setup('a b c d')
print '\\bm{a|(b*c)} =',a|(b*c)
print '\\bm{a|(b^c)} =',a|(b^c)
print '\\bm{a|(b^c^d)} =',a|(b^c^d)
print '\\bm{a|(b^c)+c|(a^b)+b|(c^a)} =',(a|(b^c))+(c|(a^b))+(b|(c^a))
print '\\bm{a*(b^c)-b*(a^c)+c*(a^b)} =',a*(b^c)-b*(a^c)+c*(a^b)
print '\\bm{a*(b^c^d)-b*(a^c^d)+c*(a^b^d)-d*(a^b^c)} =',a*(b^c^d)-b*(a^c^d)+c*(a^b^d)-d*(a^b^c)
print '\\bm{(a^b)|(c^d)} =',(a^b)|(c^d)
print '\\bm{((a^b)|c)|d} =',((a^b)|c)|d
print '\\bm{(a^b)\\times (c^d)} =',Com(a^b,c^d)

xdvi()
```

The preceeding code block also demonstrates the mapping of *, ^, and | to appropriate latex symbols.

Note: The $\times$ symbol is the commutator product of two multivectors, $A \times B = (AB - BA)/2$.

$$
\begin{aligned}
\boldsymbol{a}\cdot(\boldsymbol{bc}) &= -(a\cdot c)\boldsymbol{b} + (a\cdot b)\boldsymbol{c}\\
\boldsymbol{a}\cdot(\boldsymbol{b}\wedge\boldsymbol{c}) &= -(a\cdot c)\boldsymbol{b} + (a\cdot b)\boldsymbol{c}\\
\boldsymbol{a}\cdot(\boldsymbol{b}\wedge\boldsymbol{c}\wedge\boldsymbol{d}) &= (a\cdot d)\boldsymbol{b}\wedge\boldsymbol{c} - (a\cdot c)\boldsymbol{b}\wedge\boldsymbol{d} + (a\cdot b)\boldsymbol{c}\wedge\boldsymbol{d}\\
\boldsymbol{a}\cdot(\boldsymbol{b}\wedge\boldsymbol{c}) + \boldsymbol{c}\cdot(\boldsymbol{a}\wedge\boldsymbol{b}) + \boldsymbol{b}\cdot(\boldsymbol{c}\wedge\boldsymbol{a}) &= 0\\
\boldsymbol{a}(\boldsymbol{b}\wedge\boldsymbol{c}) - \boldsymbol{b}(\boldsymbol{a}\wedge\boldsymbol{c}) + \boldsymbol{c}(\boldsymbol{a}\wedge\boldsymbol{b}) &= 3\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{c}\\
\boldsymbol{a}(\boldsymbol{b}\wedge\boldsymbol{c}\wedge\boldsymbol{d}) - \boldsymbol{b}(\boldsymbol{a}\wedge\boldsymbol{c}\wedge\boldsymbol{d}) + \boldsymbol{c}(\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{d}) - \boldsymbol{d}(\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{c}) &= 4\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{c}\wedge\boldsymbol{d}\\
(\boldsymbol{a}\wedge\boldsymbol{b})\cdot(\boldsymbol{c}\wedge\boldsymbol{d}) &= -(a\cdot c)(b\cdot d) + (a\cdot d)(b\cdot c)\\
((\boldsymbol{a}\wedge\boldsymbol{b})\cdot\boldsymbol{c})\cdot\boldsymbol{d} &= -(a\cdot c)(b\cdot d) + (a\cdot d)(b\cdot c)\\
(\boldsymbol{a}\wedge\boldsymbol{b})\times(\boldsymbol{c}\wedge\boldsymbol{d}) &= -(b\cdot d)\boldsymbol{a}\wedge\boldsymbol{c} + (b\cdot c)\boldsymbol{a}\wedge\boldsymbol{d} + (a\cdot d)\boldsymbol{b}\wedge\boldsymbol{c} - (a\cdot c)\boldsymbol{b}\wedge\boldsymbol{d}
\end{aligned}
$$

Reciprocal Frame

The reciprocal frame of vectors with respect to the basis vectors is required for the evaluation of the geometric dervative. The following example demonstrates that for the case of an arbitrary 3-dimensional Euclidian basis the reciprocal basis vectors are correctly calculated.

```
from GAPrint import xdvi
from GA import *
Format()

metric = '1 # #,'+ \
         '# 1 #,'+ \
         '# # 1,'

(e1,e2,e3) = MV.setup('e1 e2 e3',metric)

E = e1^e2^e3
Esq = (E*E).scalar()
print 'E =',E
print '%E^{2} =',Esq
Esq_inv = 1/Esq

E1 = (e2^e3)*E
E2 = (-1)*(e1^e3)*E
E3 = (e1^e2)*E

print 'E1 = (e2^e3)*E =',E1
print 'E2 =-(e1^e3)*E =',E2
print 'E3 = (e1^e2)*E =',E3

print 'E1|e2 =',(E1|e2).expand()
print 'E1|e3 =',(E1|e3).expand()
print 'E2|e1 =',(E2|e1).expand()
print 'E2|e3 =',(E2|e3).expand()
print 'E3|e1 =',(E3|e1).expand()
print 'E3|e2 =',(E3|e2).expand()
w = ((E1|e1).expand()).scalar()
Esq = expand(Esq)
print '%(E1\\cdot e1)/E^{2} =',simplify(w/Esq)
w = ((E2|e2).expand()).scalar()
print '%(E2\\cdot e2)/E^{2} =',simplify(w/Esq)
w = ((E3|e3).expand()).scalar()
print '%(E3\\cdot e3)/E^{2} =',simplify(w/Esq)

xdvi()
```

The preceeding code also demonstrated the use of the % directive in printing a string so that ^ is treated literally and not translated to *wedge*. Note that `'%E^{2} ='` is printed as $E^2 =$ and not as $E \wedge 2 =$.

$$E = \boldsymbol{e_1} \wedge \boldsymbol{e_2} \wedge \boldsymbol{e_3}$$

$$E^2 = (e_1 \cdot e_2)^2 - 2(e_1 \cdot e_2)(e_1 \cdot e_3)(e_2 \cdot e_3) + (e_1 \cdot e_3)^2 + (e_2 \cdot e_3)^2 - 1$$

$$E1 = (e2 \wedge e3)E = \left((e_2 \cdot e_3)^2 - 1\right)\boldsymbol{e_1} + ((e_1 \cdot e_2) - (e_1 \cdot e_3)(e_2 \cdot e_3))\boldsymbol{e_2} + (-(e_1 \cdot e_2)(e_2 \cdot e_3) + (e_1 \cdot e_3))\boldsymbol{e_3}$$

$$E2 = -(e1 \wedge e3)E = ((e_1 \cdot e_2) - (e_1 \cdot e_3)(e_2 \cdot e_3))\boldsymbol{e_1} + \left((e_1 \cdot e_3)^2 - 1\right)\boldsymbol{e_2} + (-(e_1 \cdot e_2)(e_1 \cdot e_3) + (e_2 \cdot e_3))\boldsymbol{e_3}$$

$$E3 = (e1 \wedge e2)E = (-(e_1 \cdot e_2)(e_2 \cdot e_3) + (e_1 \cdot e_3))\boldsymbol{e_1} + (-(e_1 \cdot e_2)(e_1 \cdot e_3) + (e_2 \cdot e_3))\boldsymbol{e_2} + \left((e_1 \cdot e_2)^2 - 1\right)\boldsymbol{e_3}$$

$$E1 \cdot e2 = 0$$

$$E1 \cdot e3 = 0$$

$$E2 \cdot e1 = 0$$

$$E2 \cdot e3 = 0$$

$$E3 \cdot e1 = 0$$

$$E3 \cdot e2 = 0$$

$$(E1 \cdot e1)/E^2 = 1$$

$$(E2 \cdot e2)/E^2 = 1$$

$$(E3 \cdot e3)/E^2 = 1$$

The formulas derived for $E1$, $E2$, $E3$, and E^2 could also be applied to the numerical calculations of crystal properties.

Lorentz-Transformation

A simple physics demonstation of geometric algebra is the derivation of the Lorentz-Transformation. In this demonstration a 2-dimensional Minkowski space is defined and the Lorentz-Transformation is generated from a rotation of a vector in the Minkowski space using the rotor R.

```
from sympy import symbols,sinh,cosh
from GAPrint import xdvi
from GA import *

Format()
(alpha,beta,gamma) = symbols('alpha beta gamma')
```

```
(x,t,xp,tp) = symbols("x t x' t'")
(g0,g1) = MV.setup('gamma*t|x',metric='[1,-1]')

R = cosh(alpha/2)+sinh(alpha/2)*(g0^g1)
X = t*g0+x*g1
Xp = tp*g0+xp*g1

print 'R =',R
print r"#%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} = t'\bm{\gamma'_{t}}+x'\bm{\gamma'_{x}} = R\lp t'\bm{\gamma_{t}}+x'\bm{\gamma_{x}}\rp R^{\dagger}"

Xpp = R*Xp*R.rev()
Xpp = Xpp.collect([xp,tp])
Xpp = Xpp.subs({2*sinh(alpha/2)*cosh(alpha/2):sinh(alpha),sinh(alpha/2)**2+cosh(alpha/2)**2:cosh(alpha)})
print r"%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} =",Xpp
Xpp = Xpp.subs({sinh(alpha):gamma*beta,cosh(alpha):gamma})

print r'%\f{\sinh}{\alpha} = \gamma\beta'
print r'%\f{\cosh}{\alpha} = \gamma'

print r"%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} =",Xpp.collect(gamma)

xdvi()
```

The preceeding code also demonstrates how to use the sympy *subs* functions to perform the hyperbolic half angle transformation. The code also shows the use of both the # and % directives in the text string `r"#%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} = t'\bm{\gamma'_{t}}+x'\bm{\gamma'_{x}} = R\lp t'\bm{\gamma_{t}}+x'\bm{\gamma_{x}}\rp R^{\dagger}"`. Both the # and % are needed in this text string for two reasons. First, the text string contains an = sign. The latex preprocessor uses this a key to combine the text string with a sympy expression to be printed after the text string. The # is required to inform the preprocessor that there is no sympy expression to follow. Second, the % is requires to inform the preprocessor that the text string is to be displayed in latex math mode and not in text mode (if # is present the default latex mode is text mode unless overridden by the % directive).

$$R = \cosh\left(\frac{1}{2}\alpha\right) + \sinh\left(\frac{1}{2}\alpha\right)\boldsymbol{\gamma}_t \wedge \boldsymbol{\gamma}_x$$

$$t\boldsymbol{\gamma}_t + x\boldsymbol{\gamma}_x = t'\boldsymbol{\gamma}'_t + x'\boldsymbol{\gamma}'_x = R\left(t'\boldsymbol{\gamma}_t + x'\boldsymbol{\gamma}_x\right)R^{\dagger}$$

$$t\boldsymbol{\gamma}_t + x\boldsymbol{\gamma}_x = \left(t'\cosh(\alpha) - x'\sinh(\alpha)\right)\boldsymbol{\gamma}_t + \left(-t'\sinh(\alpha) + x'\cosh(\alpha)\right)\boldsymbol{\gamma}_x$$

$$\sinh(\alpha) = \gamma\beta$$

$$\cosh(\alpha) = \gamma$$

$$t\boldsymbol{\gamma}_t + x\boldsymbol{\gamma}_x = \left(\gamma\left(-\beta x' + t'\right)\right)\boldsymbol{\gamma}_t + \left(\gamma\left(-\beta t' + x'\right)\right)\boldsymbol{\gamma}_x$$

Calculus

Derivatives in Spherical Coordinates

The following code shows how to use *GA* to use spherical coordinates. The gradient of a scalar function, f, the divergence and curl of a vector function, A, and the exterior derivative (curl) of a bivector function, B are calculated. Note that to get the standard curl of a 3-dimension function the result is multiplied by $-I$ the negative of the pseudoscalar.

Note: In geometric calculus the operator ∇^2 is well defined on its own as the geometic derivative of the geometric derivative. However, if needed we have for the vector function A the relations (since $\nabla \cdot A$ is a scalar it's curl is equal to it's geometric derivative and it's divergence is zero) -

$$\begin{aligned}
\nabla A &= \nabla \wedge A + \nabla \cdot A \\
\nabla^2 A &= \nabla(\nabla \wedge A) + \nabla(\nabla \cdot A) \\
\nabla^2 A &= \nabla \wedge (\nabla \wedge A) + \nabla \cdot (\nabla \wedge A) + \nabla \wedge (\nabla \cdot A) + \nabla \cdot (\nabla \cdot A) \\
\nabla^2 A &= \nabla \wedge (\nabla \wedge A) + (\nabla \cdot \nabla)A - \nabla(\nabla \cdot A) + \nabla(\nabla \cdot A) \\
\nabla^2 A &= \nabla \wedge \nabla \wedge A + (\nabla \cdot \nabla)A
\end{aligned}$$

In the derivation we have used that $\nabla \cdot (\nabla \wedge A) = (\nabla \cdot \nabla)A - \nabla(\nabla \cdot A)$ which is implicit in the second *BAC-CAB* formula. No parenthesis is needed for the geometric curl of the curl (exterior derivative of exterior derivative) since the $\wedge$ operation is associative unlike the vector curl operator and $\nabla \cdot \nabla$ is the usual Laplacian operator.

```
from sympy import sin,cos
from GAPrint import xdvi
from GA import *
Format()

X = (r,th,phi) = symbols('r theta phi')
curv = [[r*cos(phi)*sin(th),r*sin(phi)*sin(th),r*cos(th)],[1,r,r*sin(th)]]
(er,eth,ephi,grad) = MV.setup('e_r e_theta e_phi',metric='[1,1,1]',coords=X,curv=curv)

f = MV('f','scalar',fct=True)
A = MV('A','vector',fct=True)
B = MV('B','grade2',fct=True)

print 'A =',A
print 'B =',B

print 'grad*f =',grad*f
print 'grad|A =',grad|A
print '-I*(grad^A) =',-MV.I*(grad^A)
print 'grad^B =',grad^B

xdvi()
```

Results of code

$$\begin{aligned}
A &= A^{r}\boldsymbol{e}_{\boldsymbol{r}} + A^{\theta}\boldsymbol{e}_{\boldsymbol{\theta}} + A^{\phi}\boldsymbol{e}_{\boldsymbol{\phi}} \\
B &= B^{r\theta}\boldsymbol{e}_{\boldsymbol{r}} \wedge \boldsymbol{e}_{\boldsymbol{\theta}} + B^{r\phi}\boldsymbol{e}_{\boldsymbol{r}} \wedge \boldsymbol{e}_{\boldsymbol{\phi}} + B^{\theta\phi}\boldsymbol{e}_{\boldsymbol{\theta}} \wedge \boldsymbol{e}_{\boldsymbol{\phi}} \\
\boldsymbol{\nabla} f &= \partial_{r} f\boldsymbol{e}_{\boldsymbol{r}} + \frac{\partial_{\theta} f}{r}\boldsymbol{e}_{\boldsymbol{\theta}} + \frac{\partial_{\phi} f}{r \sin{(\theta)}}\boldsymbol{e}_{\boldsymbol{\phi}} \\
\boldsymbol{\nabla} \cdot A &= \partial_{r} A^{r} + \frac{A^{\theta}}{r \tan{(\theta)}} + 2\frac{A^{r}}{r} + \frac{\partial_{\theta} A^{\theta}}{r} + \frac{\partial_{\phi} A^{\phi}}{r \sin{(\theta)}} \\
-I(\boldsymbol{\nabla} \wedge A) &= \left(\frac{A^{\phi} \cos{(\theta)} + \sin{(\theta)}\partial_{\theta} A^{\phi} - \partial_{\phi} A^{\theta}}{r \sin{(\theta)}}\right)\boldsymbol{e}_{\boldsymbol{r}} + \left(-\partial_{r} A^{\phi} - \frac{A^{\phi}}{r} + \frac{\partial_{\phi} A^{r}}{r \sin{(\theta)}}\right)\boldsymbol{e}_{\boldsymbol{\theta}} + \left(\frac{r\partial_{r} A^{\theta} + A^{\theta} - \partial_{\theta} A^{r}}{r}\right)\boldsymbol{e}_{\boldsymbol{\phi}} \\
\boldsymbol{\nabla} \wedge B &= \left(\partial_{r} B^{\theta\phi} + 2\frac{B^{\theta\phi}}{r} - \frac{B^{r\phi}}{r \tan{(\theta)}} - \frac{\partial_{\theta} B^{r\phi}}{r} + \frac{\partial_{\phi} B^{r\theta}}{r \sin{(\theta)}}\right)\boldsymbol{e}_{\boldsymbol{r}} \wedge \boldsymbol{e}_{\boldsymbol{\theta}} \wedge \boldsymbol{e}_{\boldsymbol{\phi}}
\end{aligned}$$

Maxwell's Equations

The geometric algebra formulation of Maxwell's equations is deomonstrated with the formalism developed in "Geometric Algebra for Physicists" [Doran]. In this formalism the signature of the metric is $(1,-1,-1,-1)$ and the basis vectors are γ_t, γ_x, γ_y, and γ_z. The if $\boldsymbol{E}$ and $\boldsymbol{B}$ are the normal electric and magnetic field vectors the electric and magnetic bivectors are given by $E = \boldsymbol{E}\gamma_t$ and $B = \boldsymbol{B}\gamma_t$. The electromagnetic bivector is then $F = E + IB$ where $I = \gamma_t\gamma_x\gamma_y\gamma_z$ is the pesudo-scalar for the Minkowski space. Note that the electromagnetic bivector is isomorphic to the electromagnetic tensor. Then if J is the 4-current all of Maxwell's equations are given by $\boldsymbol{\nabla} F = J$. For more details see [Doran] chapter 7.

```
from sympy import symbols,sin,cos
from GAPrint import xdvi
from GA import *

Format()

vars = symbols('t x y z')
(g0,g1,g2,g3,grad) = MV.setup('gamma*t|x|y|z',metric='[1,-1,-1,-1]',coords=vars)
I = MV.I

B = MV('B','vector',fct=True)
E = MV('E','vector',fct=True)
B.set_coef(1,0,0)
E.set_coef(1,0,0)
B *= g0
E *= g0
J = MV('J','vector',fct=True)
F = E+I*B

print 'B = \\bm{B\\gamma_{t}} =',B
print 'E = \\bm{E\\gamma_{t}} =',E
print 'F = E+IB =',F
print 'J =',J
gradF = grad*F
gradF.Fmt(3,'grad*F')

print 'grad*F = J'
(gradF.grade(1)-J).Fmt(3,'%\\grade{\\nabla F}_{1} -J = 0')
(gradF.grade(3)).Fmt(3,'%\\grade{\\nabla F}_{3} = 0')

xdvi()
```

$$
\begin{aligned}
B = \boldsymbol{B}\boldsymbol{\gamma_t} &= -B^{x}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x} - B^{y}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y} - B^{z}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_z} \\
E = \boldsymbol{E}\boldsymbol{\gamma_t} &= -E^{x}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x} - E^{y}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y} - E^{z}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_z} \\
F = E + IB &= -E^{x}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x} - E^{y}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y} - E^{z}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_z} - B^{z}\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y} + B^{y}\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_z} - B^{x}\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
J &= J^{t}\boldsymbol{\gamma_t} + J^{x}\boldsymbol{\gamma_x} + J^{y}\boldsymbol{\gamma_y} + J^{z}\boldsymbol{\gamma_z} \\
\boldsymbol{\nabla} F &= \left(\partial_{x} E^{x} + \partial_{y} E^{y} + \partial_{z} E^{z}\right)\boldsymbol{\gamma_t} \\
&\quad + \left(-\partial_{z} B^{y} + \partial_{y} B^{z} - \partial_{t} E^{x}\right)\boldsymbol{\gamma_x} \\
&\quad + \left(\partial_{z} B^{x} - \partial_{x} B^{z} - \partial_{t} E^{y}\right)\boldsymbol{\gamma_y} \\
&\quad + \left(-\partial_{y} B^{x} + \partial_{x} B^{y} - \partial_{t} E^{z}\right)\boldsymbol{\gamma_z} \\
&\quad + \left(-\partial_{t} B^{z} + \partial_{y} E^{x} - \partial_{x} E^{y}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y} \\
&\quad + \left(\partial_{t} B^{y} + \partial_{z} E^{x} - \partial_{x} E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_z} \\
&\quad + \left(-\partial_{t} B^{x} + \partial_{z} E^{y} - \partial_{y} E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
&\quad + \left(\partial_{x} B^{x} + \partial_{y} B^{y} + \partial_{z} B^{z}\right)\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
\boldsymbol{\nabla} F &= J \\
\left\langle \nabla F\right\rangle_{1} - J = 0 &= \left(-J^{t} + \partial_{x} E^{x} + \partial_{y} E^{y} + \partial_{z} E^{z}\right)\boldsymbol{\gamma_t} \\
&\quad + \left(-J^{x} - \partial_{z} B^{y} + \partial_{y} B^{z} - \partial_{t} E^{x}\right)\boldsymbol{\gamma_x} \\
&\quad + \left(-J^{y} + \partial_{z} B^{x} - \partial_{x} B^{z} - \partial_{t} E^{y}\right)\boldsymbol{\gamma_y} \\
&\quad + \left(-J^{z} - \partial_{y} B^{x} + \partial_{x} B^{y} - \partial_{t} E^{z}\right)\boldsymbol{\gamma_z} \\
\left\langle \nabla F\right\rangle_{3} = 0 &= \left(-\partial_{t} B^{z} + \partial_{y} E^{x} - \partial_{x} E^{y}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y} \\
&\quad + \left(\partial_{t} B^{y} + \partial_{z} E^{x} - \partial_{x} E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_z} \\
&\quad + \left(-\partial_{t} B^{x} + \partial_{z} E^{y} - \partial_{y} E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
&\quad + \left(\partial_{x} B^{x} + \partial_{y} B^{y} + \partial_{z} B^{z}\right)\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z}
\end{aligned}
$$

Dirac Equation

In [Doran] equation 8.89 (page 283) is the geometric algebra formulation of the Dirac equation. In this equation ψ is an 8-component real spinor which is to say that it is a multivector with sacalar, bivector, and pseudo-vector components in the space-time geometric algebra (it consists only of even grade components).

```
from sympy import symbols,sin,cos
from GAPrint import xdvi
from GA import *

Format()

vars = symbols('t x y z')
(g0,g1,g2,g3,grad) = MV.setup('gamma*t|x|y|z',metric='[1,-1,-1,-1]',coords=vars)
I = MV.I
(m,e) = symbols('m e')
```

```
psi = MV('psi','spinor',fct=True)
A = MV('A','vector',fct=True)
sig_z = g3*g0
print '\\bm{A} =',A
print '\\bm{\\psi} =',psi

dirac_eq = (grad*psi)*I*sig_z-e*A*psi-m*psi*g0
dirac_eq.simplify()

dirac_eq.Fmt(3,r'\nabla \bm{\psi} I \sigma_{z}-e\bm{A}\bm{\psi}-m\bm{\psi}\gamma_{t} = 0')

xdvi()
```

The equations displayed are the partial differential equations for each component of the Dirac equation in rectangular coordinates we the driver for the equations is the 4-potential A. One utility of these equations is to setup a numerical solver for the Dirac equation.

$$\boldsymbol{A} = A^{t}\boldsymbol{\gamma_{t}} + A^{x}\boldsymbol{\gamma_{x}} + A^{y}\boldsymbol{\gamma_{y}} + A^{z}\boldsymbol{\gamma_{z}}$$

$$\boldsymbol{\psi} = \psi + \psi^{tx}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{x}} + \psi^{ty}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{y}} + \psi^{tz}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{z}} + \psi^{xy}\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{y}} + \psi^{xz}\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{z}} + \psi^{yz}\boldsymbol{\gamma_{y}}\wedge\boldsymbol{\gamma_{z}} + \psi^{txyz}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{y}}\wedge\boldsymbol{\gamma_{z}}$$

$$\begin{aligned}
\nabla\boldsymbol{\psi}I\sigma_{z} - e\boldsymbol{A}\boldsymbol{\psi} - m\boldsymbol{\psi}\gamma_{t} = 0 = {} & \left(-eA^{t}\psi - eA^{x}\psi^{tx} - eA^{y}\psi^{ty} - eA^{z}\psi^{tz} - m\psi - \partial_{y}\psi^{tx} - \partial_{z}\psi^{txyz} + \partial_{x}\psi^{ty} + \partial_{t}\psi^{xy}\right)\boldsymbol{\gamma_{t}} \\
& + \left(-eA^{t}\psi^{tx} - eA^{x}\psi - eA^{y}\psi^{xy} - eA^{z}\psi^{xz} + m\psi^{tx} + \partial_{y}\psi - \partial_{t}\psi^{ty} - \partial_{x}\psi^{xy} + \partial_{z}\psi^{yz}\right)\boldsymbol{\gamma_{x}} \\
& + \left(-eA^{t}\psi^{ty} + eA^{x}\psi^{xy} - eA^{y}\psi - eA^{z}\psi^{yz} + m\psi^{ty} - \partial_{x}\psi + \partial_{t}\psi^{tx} - \partial_{y}\psi^{xy} - \partial_{z}\psi^{xz}\right)\boldsymbol{\gamma_{y}} \\
& + \left(-eA^{t}\psi^{tz} + eA^{x}\psi^{xz} + eA^{y}\psi^{yz} - eA^{z}\psi + m\psi^{tz} + \partial_{t}\psi^{txyz} - \partial_{z}\psi^{xy} + \partial_{y}\psi^{xz} - \partial_{x}\psi^{yz}\right)\boldsymbol{\gamma_{z}} \\
& + \left(-eA^{t}\psi^{xy} + eA^{x}\psi^{ty} - eA^{y}\psi^{tx} - eA^{z}\psi^{txyz} - m\psi^{xy} - \partial_{t}\psi + \partial_{x}\psi^{tx} + \partial_{y}\psi^{ty} + \partial_{z}\psi^{tz}\right)\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{y}} \\
& + \left(-eA^{t}\psi^{xz} + eA^{x}\psi^{tz} + eA^{y}\psi^{txyz} - eA^{z}\psi^{tx} - m\psi^{xz} + \partial_{x}\psi^{txyz} + \partial_{z}\psi^{ty} - \partial_{y}\psi^{tz} - \partial_{t}\psi^{yz}\right)\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{z}} \\
& + \left(-eA^{t}\psi^{yz} - eA^{x}\psi^{txyz} + eA^{y}\psi^{tz} - eA^{z}\psi^{ty} - m\psi^{yz} - \partial_{z}\psi^{tx} + \partial_{y}\psi^{txyz} + \partial_{x}\psi^{tz} + \partial_{t}\psi^{xz}\right)\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{y}}\wedge\boldsymbol{\gamma_{z}} \\
& + \left(-eA^{t}\psi^{txyz} - eA^{x}\psi^{yz} + eA^{y}\psi^{xz} - eA^{z}\psi^{xy} + m\psi^{txyz} + \partial_{z}\psi - \partial_{t}\psi^{tz} - \partial_{x}\psi^{xz} - \partial_{y}\psi^{yz}\right)\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{y}}\wedge\boldsymbol{\gamma_{z}}
\end{aligned}$$

[Doran] *(1, 2, 3, 4, 5, 6, 7, 8, 9)* http://www.mrao.cam.ac.uk/~cjld1/pages/book.htm `Geometric Algebra for Physicists` by C. Doran and A. Lasenby, Cambridge University Press, 2003.

[Hestenes] *(1, 2, 3, 4)* http://geocalc.clas.asu.edu/html/CA_to_GC.html `Clifford Algebra to Geometric Calculus` by D.Hestenes and G. Sobczyk, Kluwer Academic Publishers, 1984.

[Macdonald] '<http://faculty.luther.edu/~macdonal>'_ `Linear and Geometric Algebra` by Alan Macdonald, http://www.amazon.com/Alan-Macdonald/e/B004MB2QJQ