

# DESIGN OF A PYTHON MODULE FOR SYMBOLIC GEOMETRIC ALGEBRA CALCULATIONS

Alan Bromborsky  
abrombo@verizon.net  
12435 Kemp Mill Road  
Silver Spring, MD 20902

March 24, 2008

## Abstract

A python module (GAsympy.py) 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 pseudometric defined by the inner products of a set of arbitrary vectors or the pseudometric can be restricted to enforce orthogonality and signature constraints on the set of vectors. The module requires the numpy and the sympy modules. A simple calculator program is included for those who do not wish to program in python.

## 1 Introduction

Several software packages for numerical geometric algebra calculations are available from Doran-Lazenby group and the Dorst group. Symbolic packages for Clifford algebra using orthogonal bases such as  $e_i e_j + e_j e_i = 2\eta_{ij}$ , where  $\eta_{ij}$  is a numeric array are available from the Doran-Lazenby group. The symbolic algebra module, GAsympy.py, developed for python does not depend on an orthogonal basis representation, but rather is generated from a set of  $n$  arbitrary symbolic vectors,  $a_1, a_2, \dots, a_n$  and a symbolic pseudo-metric tensor  $g_{ij} = a_i \cdot a_j$ .

In order not to reinvent the wheel all scalar symbolic algebra is handled by the python module (library) sympy. **The basic classes used from the sympy module are the creation of symbolic scalar symbols and numerical symbols for exact rational arithmetic:**

## Examples of sympy class usage

```
import sympy
x = sympy.Symbol('x')
half = sympy.Rational(4,8)
print x,half
x,1/2
```

as you can see in the example rational numbers are simplified. Also, if we had used 'ab' for the argument of `sympy.Symbol` then `print x` would have returned ab.

The basic geometric algebra operations will be implemented in python by defining a multivector class, MV, and overloading the class operators and defining class functions shown in Table 1. Note that the operator order precedence is

+	sum of multivectors or multivector and scalar
-	difference of multivectors or multivector and scalar
*	geometric product or multiplication by scalar
^	outer product of multivectors
	inner product of multivectors
rev()	reverse of multivector
even()	even part of multivector
odd()	odd part of multivector
project(r)	grade r part of multivector
X(ig,ib)	For a multivector X the call operator gives the symbolic coefficient of the ib base of the ig grade of the multivector. Because of the defaults in the function call X() returns the scalar part of X.

Table 1: Multivector operations for symbolicGA.py.

determined by python and is not necessarily that used by geometric algebra. Always use parenthesis in python expressions containing  $\wedge$  and/or  $|$ .

Complete documentation of GAsympy is obtained by running `pydoc GAsympy` or `pydoc -w GAsympy` (for html document) in the directory containing GAsympy.

If the reader is not familiar with python the recommended reference is **Learning Python: 3<sup>rd</sup> edition** by Mark Lutz.

The GAsympy module was implemented in python 2.5.1 on a linux machine running Kubuntu. The only required packages (in addition to the standard python installation) are numpy and sympy.

## 2 Vector Basis and Metric

The two structures that define the MV (multivector) class are the symbolic basis vectors and the symbolic pseudometric. 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 pseudometric for the geometric algebra of the basis we have defined. The default basis is the most general and is the matrix of the following symbols

$$g = \begin{bmatrix} \mathbf{a0} ** 2 & (\mathbf{a0.a1}) & (\mathbf{a0.a2}) \\ (\mathbf{a0.a1}) & \mathbf{a1} ** 2 & (\mathbf{a1.a2}) \\ (\mathbf{a0.a2}) & (\mathbf{a1.a2}) & \mathbf{a2} ** 2 \end{bmatrix} \quad (1)$$

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  $(\mathbf{a0.a1}) \neq (\mathbf{a1.a0})$ .

Note that the strings shown in equation 1 are only used when the values of  $g_{ij}$  are output (printed). In the **GA**sympy module (library) the  $g_{ij}$  symbols are stored in a static member list of the multivector class MV as the double list **MV.metric** ( $g_{ij} = \mathbf{MV.metric}[i][j]$ ).

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'
```

which is processed by setup to yield

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

Here we have specified that **n** and **nbar** are orthonal to all the **a**'s,  $\mathbf{n} ** 2 = \mathbf{nbar} ** 2 = 0$ , and  $(\mathbf{n.nbar}) = 2$ . Using # in the metric definition string just

tells the program to use the default symbol for that value. The multivector algebra is then initialized with the function call

```
MV.setup(basis,metric)
```

At this time multivector representations of the basis local to the program are instantiated. For our first example that means that the symbolic multivectors named `a0`, `a1`, and `a2` are created and made available to the programmer for future calculations.

In addition to the basis vectors the  $g_{ij}$  are also made available to the programmer with the following convention. If `a0` and `a1` are basis vectors, then their dot products are denoted by `a0sq`, `a2sq`, and `a0dota1` for use as python program variables. If you print `a0sq` the output would be `a0**2` and the output for `a0dota1` would be `(a0.a1)` as shown in equation 1. If the default value are overridden the new values are output by print. For example if  $g_{00} = 0$  then `print a0sq` would output "0."

More generally, if `metric` is not a string, but a list of lists, it is assumed that each element of `metric` is symbolic variable so that the  $g_{ij}$  could be defined as symbolic functions as well as variables. For example instead of letting  $g_{01} = (a0.a1)$  we could have  $g_{01} = \cos(\theta)$  where we use a symbolic `cos` function.

### 3 Representation and Reduction of Multivector Bases

**Python Note:** In python a list of objects is denoted with square brackets `[]`. For example a list of the integers 1, 2, and 3 would be written in python code as `[1,2,3]`. An object in a list can itself be a list (nested lists). Objects in a list containing  $n$ -objects are indexed  $0, 1, \dots, n - 1$  where if `B` is the list, `B[0]` is the first object in the list and `B[n-1]` is the last object in the list.

In our symbolic geometric algebra we assume that all multivectors of interest to us can be obtained from the symbolic bases 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  $i_1 < i_2 < \dots < i_r$  and  $r \leq n$ . We call these multivectors bases and represent them internally with the list of integers  $[i_1, i_2, \dots, i_r]$ . The bases are labeled, for the purpose of output display, with strings that are concatenations of the strings representing the basis vectors. So that in our example `[1,2]` would be labeled with the string `'a1a2'` and represents the geometric product `a1*a2`. Thus the list `[0,1,2]` represents `a0*a1*a2`. For our example the complete set of

bases and labels are shown in Table 2<sup>1</sup> Since there are  $2^n$  bases and the number

```
MV.basislabel = ['1', ['a0', 'a1', 'a2'], ['a0a1', 'a0a2', 'a1a2'],
                 ['a0a1a2']]
MV.basis      = [[], [[0], [1], [2]], [[0, 1], [0, 2], [1, 2]],
                 [[0, 1, 2]]]
```

Table 2: Multivector basis labels and internal basis representation.

of bases with equal list lengths is the same as for the grade decomposition of a dimension  $n$  geometric algebra we will call the collections of bases of equal length **psuedogrades**.

The critical operation in setting up the geometric algebra module 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. First we represent the product as the concatenation of two base lists. For example  $\mathbf{a1a2} \cdot \mathbf{a0a1}$  is represented by the list  $[1, 2] + [0, 1] = [1, 2, 0, 1]$  (In python the "+" operator for lists concatenates the lists). The representation of the product is reduced via two operations, contraction and revision. The state of the reduction is saved in two lists of equal length. The first list contains symbolic scale factors (symbol or numeric types) for the corresponding interger list representing the product of bases. If we wish to reduce  $[i_1, \dots, i_r]$  the starting point is the coefficient list  $C = [1]$  and the bases list  $B = [[i_1, \dots, i_r]]$ . We now operate on each element of the lists as follows:

---

<b>contraction</b>	Consider a basis list $B$ with element $B[j] = [i_1, \dots, i_l, i_{l+1}, \dots, i_s]$ where $i_l = i_{l+1}$ . Then the product of the $l$ and $l + 1$ terms result in a scalar and $B[j]$ is replaced by the new list representation $[i_1, \dots, i_{l-1}, i_{l+2}, \dots, i_s]$ which is of psuedo grade $r - 2$ and $C[j]$ is replaced by the symbol $g_{i_l i_l} C[j]$ .
--------------------	---

---

<b>revision</b>	Consider a basis list $B$ with element $B[j] = [i_1, \dots, i_l, i_{l+1}, \dots, i_s]$ where $i_l > i_{l+1}$ . Then the $l$ and $l + 1$ elements must be reversed to be put in normal order, but we have $a_{i_l} a_{i_{l+1}} = 2g_{i_l i_{l+1}} - a_{i_{l+1}} a_{i_l}$ (From the geometric algebra definition of the dot product of two vectors). Thus we append the list representing the reduced element $[i_1, \dots, i_{l-1}, i_{l+2}, \dots, i_s]$ , to the pseudo bases list, $B$ , and append $2g_{i_l i_{l+1}} C[j]$ to the coefficients list, then we replace $B[j]$ with $[i_1, \dots, i_{l+1}, i_l, \dots, i_s]$ and $C[j]$ with $-C[j]$ . Both lists are increased by one element if $g_{i_l i_{l+1}} \neq 0$ .
-----------------	--

---

<sup>1</sup>The empty list, [], represents the scalar 1.

These processes are repeated until every base in  $B$  is in normal (ascending) order with no repeated elements. Then the coefficients of equivalent bases are summed and the bases sorted according to pseudograde and ascending order. We now have a way of calculating the geometric product of any two bases as a symbolic linear combination of all the bases with the coefficients determined by  $g$ . The base multiplication table for our simple example of three vectors is given by (the coefficient of each pseudo base is enclosed with  $\{\}$  for clarity):

$$\begin{aligned}
(1)(1) &= 1 \\
(1)(a_0) &= a_0 \\
(1)(a_1) &= a_1 \\
(1)(a_2) &= a_2 \\
(1)(a_0a_1) &= a_0a_1 \\
(1)(a_0a_2) &= a_0a_2 \\
(1)(a_1a_2) &= a_1a_2 \\
(1)(a_0a_1a_2) &= a_0a_1a_2 \\
\\
(a_0)(1) &= a_0 \\
(a_0)(a_0) &= \{a_0^2\}1 \\
(a_0)(a_1) &= a_0a_1 \\
(a_0)(a_2) &= a_0a_2 \\
(a_0)(a_0a_1) &= \{a_0^2\}a_1 \\
(a_0)(a_0a_2) &= \{a_0^2\}a_2 \\
(a_0)(a_1a_2) &= a_0a_1a_2 \\
(a_0)(a_0a_1a_2) &= \{a_0^2\}a_1a_2 \\
\\
(a_1)(1) &= a_1 \\
(a_1)(a_0) &= \{2(a_0 \cdot a_1)\}1 - a_0a_1 \\
(a_1)(a_1) &= \{a_1^2\}1 \\
(a_1)(a_2) &= a_1a_2 \\
(a_1)(a_0a_1) &= \{-a_1^2\}a_0 + \{2(a_0 \cdot a_1)\}a_1 \\
(a_1)(a_0a_2) &= \{2(a_0 \cdot a_1)\}a_2 - a_0a_1a_2 \\
(a_1)(a_1a_2) &= \{a_1^2\}a_2 \\
(a_1)(a_0a_1a_2) &= \{-a_1^2\}a_0a_2 + \{2(a_0 \cdot a_1)\}a_1a_2 \\
\\
(a_2)(1) &= a_2 \\
(a_2)(a_0) &= \{2(a_0 \cdot a_2)\}1 - a_0a_2 \\
(a_2)(a_1) &= \{2(a_1 \cdot a_2)\}1 - a_1a_2 \\
(a_2)(a_2) &= \{a_2^2\}1 \\
(a_2)(a_0a_1) &= \{-2(a_1 \cdot a_2)\}a_0 + \{2(a_0 \cdot a_2)\}a_1 + a_0a_1a_2 \\
(a_2)(a_0a_2) &= \{-a_2^2\}a_0 + \{2(a_0 \cdot a_2)\}a_2 \\
(a_2)(a_1a_2) &= \{-a_2^2\}a_1 + \{2(a_1 \cdot a_2)\}a_2 \\
(a_2)(a_0a_1a_2) &= \{a_2^2\}a_0a_1 + \{-2(a_1 \cdot a_2)\}a_0a_2 + \{2(a_0 \cdot a_2)\}a_1a_2 \\
\\
(a_0a_1)(1) &= a_0a_1 \\
(a_0a_1)(a_0) &= \{2(a_0 \cdot a_1)\}a_0 + \{-a_0^2\}a_1 \\
(a_0a_1)(a_1) &= \{a_1^2\}a_0 \\
(a_0a_1)(a_2) &= a_0a_1a_2 \\
(a_0a_1)(a_0a_1) &= \{-a_0^2a_1^2\}1 + \{2(a_0 \cdot a_1)\}a_0a_1
\end{aligned}$$

```

(a0a1)(a0a2) = {2*(a0.a1)}a0a2+{-a0**2}a1a2
(a0a1)(a1a2) = {a1**2}a0a2
(a0a1)(a0a1a2) = {-a0**2*a1**2}a2+{2*(a0.a1)}a0a1a2

(a0a2)(1) = a0a2
(a0a2)(a0) = {2*(a0.a2)}a0+{-a0**2}a2
(a0a2)(a1) = {2*(a1.a2)}a0-a0a1a2
(a0a2)(a2) = {a2**2}a0
(a0a2)(a0a1) = {-2*a0**2*(a1.a2)}1+{2*(a0.a2)}a0a1+{a0**2}a1a2
(a0a2)(a0a2) = {-a0**2*a2**2}1+{2*(a0.a2)}a0a2
(a0a2)(a1a2) = {-a2**2}a0a1+{2*(a1.a2)}a0a2
(a0a2)(a0a1a2) = {a0**2*a2**2}a1+{-2*a0**2*(a1.a2)}a2+{2*(a0.a2)}a0a1a2

(a1a2)(1) = a1a2
(a1a2)(a0) = {2*(a0.a2)}a1+{-2*(a0.a1)}a2+a0a1a2
(a1a2)(a1) = {2*(a1.a2)}a1+{-a1**2}a2
(a1a2)(a2) = {a2**2}a1
(a1a2)(a0a1) = {2*a1**2*(a0.a2)-4*(a0.a1)*(a1.a2)}1+{2*(a1.a2)}a0a1+{-a1**2}a0a2
+{2*(a0.a1)}a1a2
(a1a2)(a0a2) = {-2*a2**2*(a0.a1)}1+{a2**2}a0a1+{2*(a0.a2)}a1a2
(a1a2)(a1a2) = {-a1**2*a2**2}1+{2*(a1.a2)}a1a2
(a1a2)(a0a1a2) = {-a1**2*a2**2}a0+{2*a2**2*(a0.a1)}a1+{2*a1**2*(a0.a2)
-4*(a0.a1)*(a1.a2)}a2+{2*(a1.a2)}a0a1a2

(a0a1a2)(1) = a0a1a2
(a0a1a2)(a0) = {2*(a0.a2)}a0a1+{-2*(a0.a1)}a0a2+{a0**2}a1a2
(a0a1a2)(a1) = {2*(a1.a2)}a0a1+{-a1**2}a0a2
(a0a1a2)(a2) = {a2**2}a0a1
(a0a1a2)(a0a1) = {2*a1**2*(a0.a2)-4*(a0.a1)*(a1.a2)}a0+{2*a0**2*(a1.a2)}a1
+{-a0**2*a1**2}a2+{2*(a0.a1)}a0a1a2
(a0a1a2)(a0a2) = {-2*a2**2*(a0.a1)}a0+{a0**2*a2**2}a1+{2*(a0.a2)}a0a1a2
(a0a1a2)(a1a2) = {-a1**2*a2**2}a0+{2*(a1.a2)}a0a1a2
(a0a1a2)(a0a1a2) = {-a0**2*a1**2*a2**2}1+{2*a2**2*(a0.a1)}a0a1+{2*a1**2*(a0.a2)
-4*(a0.a1)*(a1.a2)}a0a2+{2*a0**2*(a1.a2)}a1a2

```

## 4 Base Representation of Multivectors

**Python Note:** In GAsympy symbolic multivectors are instantiated as the python class MV. This means that associated with each instance of a particular multivector A is a set of data which defines that particular multivector and a set of operations (functions such as projection and reversal and operations such as addition, subtraction, geometric product, inner product, and outer product) defined by the class. For a particular A most of the data related to A is stored in the list of arrays A.mv. In addition there is a flag, A.bladeflg, that would indicate if A is currently a base or blade representation (see section 5) and data structures for any other required information.

In terms of the bases defined an arbitrary multivector can be represented as a list of arrays (we use the numpy python module to implement arrays). If we have  $n$  basis vectors we initialize the list `self.mv = [0,0,...,0]` with  $n + 1$  zeros. Each zero is a placeholder for an array of python objects (in this case the objects will be sympy symbol objects). If `self.mv[r] = numpy.array([list of symbol objects])` each entry in the `numpy.array` will be a coefficient of the corresponding psuedo base. `self.mv[r] = 0` indicates that the coefficients of every base of psuedo grade  $r$  are 0. The length of the array `self.mv[r]` is  $\binom{n}{r}$  the binomial coefficient. For example the psuedo basis vector `a1` would be represented as a multivector by the list:

```
a1.mv = [0,numpy.array([numeric(0),numeric(1),numeric(0)]),0,0]
```

and `a0a1a2` by:

```
a0a1a2.mv = [0,0,0,numpy.array([numeric(1)])]
```

The array is stuffed with sympy numeric objects instead of python integers so that we can perform symbolically manipulate sympy expressions that consist of scalar algebraic symbols and exact rational numbers which sympy can also represent.

The `numpy.array` is used because operations of addition, subtraction, and multiplication by an object are defined for the array if they are defined for the objects making up the array, which they are by sympy. We call this representation a base type because the `r` index is not a grade index since the bases we are using are not blades. In a blade representation the structure would be identical, but the bases would be replaced by blades and `self.mv[r]` would represent the `r` grade components of the multivector. The first use of the base representation is to store the results of the multiplication tabel for the bases in the class variable `MV.mtabel`. This variable is a group of nested lists so that the geometric product of the `igrade` and `ibase` with the `jgrade` and `jbase` is `MV.mtabel[igrade][ibase][jgrade][jbase]`. We can then use this table to calculate the geometric product of any two multivectors.

## 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

$$A_r \wedge b = \frac{1}{2} (A_r b - (-1)^r b A_r), \quad (3)$$

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. The important thing to notice about Table 3 is that it is a triangular (lower triangular) system of equations so that using a simple

```

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.

```

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 bases blades.

back substitution algorithm we can solve for the psuedo bases in terms of the blades giving Table 4. 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 zero for a base representation and 1 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.

## 6 Outer and Inner Product

**Geometric Algebra Note:** 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

$$A = \sum_{r=0}^n A_r \quad (4)$$

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

$$A_r B_s = \langle A_r B_s \rangle_{|r-s|} + \langle A_r B_s \rangle_{|r-s|+2} + \cdots + \langle A_r B_s \rangle_{r+s} \quad (5)$$

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

$$A_r \cdot B_s = \langle A_r B_s \rangle_{|r-s|} \quad (6)$$

$$A_r \wedge B_s = \langle A_r B_s \rangle_{r+s} \quad (7)$$

and

$$A \cdot B = \sum_{r,s} A_r \cdot B_s \quad (8)$$

$$A \wedge B = \sum_{r,s} A_r \wedge B_s \quad (9)$$

The MV class function for the outer product of the multivectors `mv1` and `mv2` is

```
def outer_product(mv1,mv2):
    product = MV()
    product.bladeflg = 1
    mv1.convert_to_blades()
    mv2.convert_to_blades()
    for igrade1 in MV.n1rg:
        if not isint(mv1.mv[igrade1]):
            pg1 = mv1.project(igrade1)
            for igrade2 in MV.n1rg:
                igrade = igrade1+igrade2
                if igrade <= MV.n:
                    if not isint(mv2.mv[igrade2]):
                        pg2 = mv2.project(igrade2)
                        pg1pg2 = pg1*pg2
                        product.add_in_place(pg1pg2.project(igrade))
    return(product)
outer_product = staticmethod(outer_product)
```

In the MV class we have overloaded the  $\wedge$  operator so that instead of calling the function we can write `mv1^mv2`. Due to the precedence rules for python we should **always** enclose outer and inner products in parenthesis. The steps for calculating the outer product are:

1. Convert `mv1` and `mv2` to blade representation if they are not already in that form.

2. Project and loop through each grade `mv1.mv[i1]` and `mv2.mv[i2]`.
3. Calculate the geometric product `pg1*pg2`.
4. Project the `i1+i2` grade from `pg1*pg2`.
5. Accumulate the results for each pair of grades in the input multivectors.

For the inner product of the multivectors `mv1` and `mv2` the `MV` class function is

```
def inner_product(mv1,mv2):
    product = MV()
    product.bladeflg = 1
    mv1.convert_to_blades()
    mv2.convert_to_blades()
    for igrade1 in range(1,MV.n1):
        if not isint(mv1.mv[igrade1]):
            pg1 = mv1.project(igrade1)
            for igrade2 in range(1,MV.n1):
                igrade = abs(igrade1-igrade2)
                if not isint(mv2.mv[igrade2]):
                    pg2 = mv2.project(igrade2)
                    pg1pg2 = pg1*pg2
                    product.add_in_place(pg1pg2.project(igrade))
    return(product)
inner_product = staticmethod(inner_product)
```

In the `MV` class we have overloaded the `|` operator so that instead of calling the function we can write `mv1|mv2`. The inner product is calculated the same way as the outer product except that in step 4, `i1+i2` is replaced by `abs(i1-i2)`.

## 7 Examples of GAsympy in Action

We now give three examples of symbolicGA in use (all are from **Geometric Algebra for Physicists** by Doran and Lazenby). The complete python code for all three examples is given in the file `test_symbolicGA.py` and the results in `test_symbolicGA.out`.

### 7.1 Derive Non-Euclidian Distance

We shall derive the formula for calculating the distance in hyperbolic space from chapter 10 of **Geometric Algebra for Physicists** by Doran and Lazenby. The equations we must solve (pages 373-374) are

$$B = (X \wedge Y \wedge e)e = Le, \text{ where } X^2 = Y^2 = 0 \text{ and } e^2 = 1 \quad (10)$$

$$\hat{B} = \frac{B}{\sqrt{B^2}} \quad (11)$$

$$Y = e^{\frac{\alpha \hat{B}}{2}} X e^{-\frac{\alpha \hat{B}}{2}} \quad (12)$$

To solve equation 12 note that  $Y \wedge Y = 0$  so that equation 12 is satisfied if,

$$\left( e^{\frac{\alpha \hat{B}}{2}} X e^{-\frac{\alpha \hat{B}}{2}} \right) \cdot Y = 0 \quad (13)$$

and we use

$$e^{\frac{\alpha \hat{B}}{2}} = \cosh\left(\frac{\alpha}{2}\right) + \sinh\left(\frac{\alpha}{2}\right) \hat{B} \quad (14)$$

to solve for  $\alpha$  as a function of  $X$  and  $Y$ .

The python code used to solve equation 13 is shown below. Note the comments (all lines between ""'s) that explain the inputs and outputs.

```
print 'Example: non-euclidian distance calculation'

metric = '0 # #,' + \
'# 0 #,' + \
'# # 1,'

MV.setup('X Y e',metric,debug=0)
MV.set_str_format(1)

"""
X and Y are conformal mappings of two vectors on the Poincare disk
"""

L = X^Y^e

"""
B is the bivector generator of translations on the circle defined by L
"""

B = L*e
Bsq = (B*B)()
print 'L = X^Y^e is a non-euclidian line'
print 'B = L*e =',B
print 'B^2 =',Bsq
print 'L^2 =',(L*L)()
```

```

#make_scalars('s c Binv')
make_symbols('s c Binv M S C alpha')
"""
s = sinh(alpha/2)
c = cosh(alpha/2)
Binv = 1/sqrt(B*B) It can be shown that B*B > 0
"""

Bhat = Binv*B # Normalize translation generator
R = c+s*Bhat # Rotor R = exp(alpha*Bhat/2)
print 's = sinh(alpha/2) and c = cosh(alpha/2)'
print 'R = exp(alpha*B/(2*|B|)) =',R
Z = R*X*R.rev()
Z.expand()
Z.collect([Binv,s,c,XdotY])
print 'R*X*R.rev() =',Z
#W = Z^Y
W = Z|Y
W.expand()
W.collect([s*Binv])
print '(R*X*rev(R)).Y =',W
M = 1/Bsq
W.subs(Binv**2,M)
W.simplify()
Bmag = sympy.sqrt(XdotY**2-2*XdotY*Xdote*Ydote)
W.collect([Binv*c*s,XdotY])

"""
Coefficients of W blades must be reduced via hyperbolic trig
substitutions to yield solution for alpha.
"""
W.subs(2*XdotY**2-4*XdotY*Xdote*Ydote,2/(Binv**2))
W.subs(2*c*s,S)
W.subs(c**2,(C+1)/2)
W.subs(s**2,(C-1)/2)
W.simplify()
W.subs(1/Binv,Bmag)
W = W()
print '(R*X*R.rev()).Y =',W
nl = '\n'
Wd = collect(W,[C,S],evaluate=False)
lhs = Wd[ONE]+Wd[C]*C
rhs = -Wd[S]*S
lhs = lhs**2
rhs = rhs**2
W = (lhs-rhs).expand()

```

```

W = (W.subs(S**2,C**2-1)).expand()
W = collect(W,[C**2,C],evaluate=False)
a = W[C**2]
b = W[abs(C)]
c = W[ONE]
D = (b**2-4*a*c).expand()
print 'Setting to 0 and solving for C gives:'
print 'Discriminant D = b^2-4*a*c =',D
C = (-b/(2*a)).expand()
print 'C = cosh(alpha) = -b/(2*a) =',C

```

Note that  $B_{inv}$  is  $|B|^{-1}$ . The critical outputs are the expressions for  $B^2$  and  $(R*X*R.rev())|Y$ . Then we solve for  $(R*X*R.rev())|Y = 0$ . Most of the code is reducing  $(R*X*R.rev())|Y = 0$  to a quadratic equation in  $\cosh(\alpha)$  which we can solve. The output of the code is:

```

example: non-euclidian distance calculation
L = X^Y^e is a non-euclidian line
B = L*e = X^Y
+{-(Y.e)}X^e
+{(X.e)}Y^e

B^2 = (X.Y)**2 - 2*(X.Y)*(X.e)*(Y.e)
L^2 = (X.Y)**2 - 2*(X.Y)*(X.e)*(Y.e)
s = sinh(alpha/2) and c = cosh(alpha/2)
R = exp(alpha*B/(2*|B|)) = {c}1
+{Binv*s}X^Y
+{-(Y.e)*Binv*s}X^e
+{(X.e)*Binv*s}Y^e

R*X*R.rev() = {c**2 + Binv*(2*(X.Y)*c*s - 2*(X.e)*(Y.e)*c*s)
+ Binv**2*((X.Y)**2*s**2 - 2*(X.Y)*(X.e)*(Y.e)*s**2)}X
+{2*Binv*c*s*(X.e)**2}Y
+{Binv**2*(-2*(X.e)*(X.Y)**2*s**2 + 4*(X.Y)*(Y.e)*(X.e)**2*s**2)
- 2*(X.Y)*(X.e)*Binv*c*s}e

(R*X*rev(R)).Y = {(X.Y)*c**2 + Binv*s*(2*c*(X.Y)**2 - 4*(X.Y)*(X.e)*(Y.e)*c)
+ Binv**2*s**2*((X.Y)**3 - 4*(X.e)*(Y.e)*(X.Y)**2
+ 4*(X.Y)*(X.e)**2*(Y.e)**2)}1

(R*X*R.rev()).Y = (X.Y)*C+(X.e)*(Y.e)+S*((X.Y)**2-2*(X.Y)*(X.e)*(Y.e))**(1/2)-(X.e)*(Y.e)
Setting to 0 and solving for C gives:
Discriminant D = b^2-4*a*c = 0
C = cosh(alpha) = -b/(2*a) = 1 - (X.Y)/(X.e)/(Y.e)

```

Due to the way sympy processes the output symbolic manipulations  $1 - (X.Y)/(X.e)/(Y.e)$

=  $1 - (X \cdot Y) / ((X \cdot e)(Y \cdot e))$ . Using the value calculated for  $B^2$  (the quantity in the radical is always positive for any point on the Poincare disk) we have

$$|B| = \sqrt{(X \cdot Y)^2 - 2(X \cdot Y)(X \cdot e)(Y \cdot e)} \quad (15)$$

The equation for  $X|Y = 0$  is then

$$(X \cdot Y - (X \cdot e)(Y \cdot e)) \cosh(\alpha) + (X \cdot e)(Y \cdot e) + (X \cdot Y)(X \cdot e)(Y \cdot e)|B| \sinh(\alpha) = 0 \quad (16)$$

Equation 16 is identical to 10.161 in Doran and Lazenby. If one employs the standard hyperbolic trig identities we get a quadratic equation in  $\cosh(\alpha)$  with solution

$$\cosh(\alpha) = 1 - \frac{X \cdot Y}{(X \cdot e)(Y \cdot e)} \quad (17)$$

which is the equation in Doran and Lazenby without the intermediate geometric algebra manipulations in their equations 10.156 through 10.161.

The biggest problem in using GAsym in this case is having enough understanding of sympy to be able to simplify the symbolic coefficient expansions so that we can solve for  $\cosh(\alpha)$ . The easiest thing to do was to use the metric tensor to enforce the required auxiliary conditions on  $X$ ,  $Y$ , and  $e$ .

## 7.2 Conformal Geometry

We shall show that blades in conformal space represent basis geometric shapes such as circles, lines, spheres, and planes. In chapter 10 of **Geometric Algebra for Physicists** by Doran and Lazenby it is shown that points in euclidian space can be represented by rays (null vectors) in a space with two appended dimensions. If basis vectors for the appended dimensions are  $e$  and  $\bar{e}$  where  $e^2 = -\bar{e}^2 = 1$  we define the null basis vectors  $n = e + \bar{e}$  and  $\bar{n} = e - \bar{e}$  and have from the definitions that  $n^2 = \bar{n}^2 = 0$  and  $n \cdot \bar{n} = 2$ . A vector  $x$  in the euclidian space is mapped into a ray in the conformal space by

$$X = F(x) = \frac{1}{2}(x^2 n + 2x - \bar{n}) \quad (18)$$

Now let  $A$ ,  $B$ ,  $C$ , and  $D$  be rays representing fixed points in a 3-D euclidian space and  $X$  representing the variable point  $x = x_1 e_1 + x_2 e_2 + x_3 e_3$  via the mapping in equation 18. Then the basis geometric shapes are represented by the blade equations:

Line	-	$(A \wedge B \wedge n) \wedge X = 0$
Circle	-	$(A \wedge B \wedge C) \wedge X = 0$
Plane	-	$(A \wedge B \wedge C \wedge n) \wedge X = 0$
Sphere	-	$(A \wedge B \wedge C \wedge D) \wedge X = 0$

The python code demonstrating this is shown below. The points on the circle and sphere are selected so that both shapes have centers at the origin and the form of the equation in  $x_1$ ,  $x_2$ , and  $x_3$  are obvious.

```
def F(x):
    Fx = HALF*((x*x)*n+2*x-nbar)
    return(Fx)

def make_vector(a,n = 3):
    if type(a) == types.StringType:
        sym_str = ''
        for i in range(n):
            sym_str += a+str(i)+' '
        sym_lst = make_symbols(sym_str)
        sym_lst.append(ZERO)
        sym_lst.append(ZERO)
        a = MV(sym_lst,'vector')
    return(F(a))

print '\n\n\nExample: Conformal representations of circles, lines,\n'+\
      ' spheres, and planes'

metric = '1 0 0 0 0,'+ \
         '0 1 0 0 0,'+ \
         '0 0 1 0 0,'+ \
         '0 0 0 0 2,'+ \
         '0 0 0 2 0'

MV.setup('e0 e1 e2 n nbar',metric,debug=0)
MV.set_str_format(1)

#conformal representation of points
A = make_vector(e0)    # point a = (1,0,0)  A = F(a)
B = make_vector(e1)    # point b = (0,1,0)  B = F(b)
C = make_vector(-1*e0) # point c = (-1,0,0) C = F(c)
D = make_vector(e2)    # point d = (0,0,1)  D = F(d)
X = make_vector('x')
print 'a = e0, b = e1, c = -e0, and d = e2'
print 'A = F(a) = 1/2*(a*a*n+2*a-nbar), etc.'
print 'Circle through a, b, and c'
print 'Circle: A^B^C^X = 0 =',(A^B^C^X)
print 'Line through a and b'
print 'Line : A^B^n^X = 0 =',(A^B^n^X)
print 'Sphere through a, b, c, and d'
print 'Sphere: A^B^C^D^X = 0 =',(A^B^C^D^X)
```

```

print 'Plane through a, b, and d'
print 'Plane : A^B^n^D^X = 0 =',(A^B^n^D^X)

```

The output of the code is shown below. This output show that the blade equations properly encode the basic geometric shapes. The shapes shown can be translated and scaled using rotations in the conformal space. Thus the blade equations are corrent for any line, circle, plane, or sphere.

Example: Conformal representations of circles, lines, spheres, and planes

a = e0, b = e1, c = -e0, and d = e2

A = F(a) = 1/2\*(a\*a^n+2\*a-nbar), etc.

Circle through a, b, and c

Circle: A^B^C^X = 0 = {-x2}e0^e1^e2^n

+{x2}e0^e1^e2^nbar

+{-1/2+1/2\*x0\*\*2+1/2\*x2\*\*2+1/2\*x1\*\*2}e0^e1^n^nbars

Line through a and b

Line : A^B^n^X = 0 = {-x2}e0^e1^e2^n

+{-1/2+1/2\*x1+1/2\*x0}e0^e1^n^nbars

+{1/2\*x2}e0^e2^n^nbars

+{-1/2\*x2}e1^e2^n^nbars

Sphere through a, b, c, and d

Sphere: A^B^C^D^X = 0 = {1/2-1/2\*x0\*\*2-1/2\*x2\*\*2-1/2\*x1\*\*2}e0^e1^e2^n^nbars

Plane through a, b, and d

Plane : A^B^n^D^X = 0 = {1/2-1/2\*x1-1/2\*x0-1/2\*x2}e0^e1^e2^n^nbars

### 7.3 Reciprocal Frames

We shall show that the set of reciprocal vectors with respect to an arbitrary basis are correctly calculated according to **Geometric Algebra for Physicists** by Doran and Lazenby, chapter 4, page 100. The only simplification we make is that our basis consists of unit vectors  $e_1^2 = e_2^2 = e_3^2 = 1$ , otherwise  $e_1$ ,  $e_2$ , and  $e_3$  are arbitrary. We program formula 4.94, the program implementing this is shown below. In the program we have  $e^j = E_j$  and  $E_n = E$  for formulas 4.92 and 4.94 in the reference.

```

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

```

```

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

```

```

print 'Example: Reciprocal Frames e1, e2, and e3 unit vectors.\n\n'

```

```

E = e1^e2^e3
Esq = (E*E)()
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
w = (E1|e2)
w.collect(MV.g)
print 'E1|e2 =',w
w = (E1|e3)
w.collect(MV.g)
print 'E1|e3 =',w
w = (E2|e1)
w.collect(MV.g)
print 'E2|e1 =',w
w = (E2|e3)
w.collect(MV.g)
print 'E2|e3 =',w
w = (E3|e1)
w.collect(MV.g)
print 'E3|e1 =',w
w = (E3|e2)
w.collect(MV.g)
print 'E3|e2 =',w
w = (E1|e1)
w = expand(w())
Esq = expand(Esq)
print '(E1|e1)/E^2 =',w/Esq
w = (E2|e2)
w = expand(w())
print '(E2|e2)/E^2 =',w/Esq
w = (E3|e3)
w = expand(w())
print '(E3|e3)/E^2 =',w/Esq

```

The output of the program is below and shows that  $e^i \cdot e_j = \delta_j^i$ , where  $\delta_j^i$  is the Kronecker delta function (one if  $i = j$ , zero if  $i \neq j$ ). The output is correct

Example: Reciprocal Frames  $e_1$ ,  $e_2$ , and  $e_3$  unit vectors.

```

E = e1^e2^e3
E^2 = -1+(e1.e3)**2+(e2.e3)**2-2*(e1.e2)*(e2.e3)*(e1.e3)+(e1.e2)**2

```

```

E1 = (e2^e3)*E = {-1+(e2.e3)**2}e1
                +{(e1.e2)-(e2.e3)*(e1.e3)}e2
                +{-(e1.e2)*(e2.e3)+(e1.e3)}e3
E2 = -(e1^e3)*E = {(e1.e2)-(e2.e3)*(e1.e3)}e1
                +{-1+(e1.e3)**2}e2
                +{-(e1.e2)*(e1.e3)+(e2.e3)}e3
E3 = (e1^e2)*E = {-(e1.e2)*(e2.e3)+(e1.e3)}e1
                +{-(e1.e2)*(e1.e3)+(e2.e3)}e2
                +{-1+(e1.e2)**2}e3

E1|e2 = 0
E1|e3 = 0
E2|e1 = 0
E2|e3 = 0
E3|e1 = 0
E3|e2 = 0
(E1|e1)/E^2 = 1
(E2|e2)/E^2 = 1
(E3|e3)/E^2 = 1

```

## 7.4 More Examples

The complete test suite (`test_symbolicGA.py`) and its output (`test_symbolicGA.out`) are part of the GAsympy documentation and include more examples than shown in this document. Documentation of all the classes and functions in GAsympy are obtained by the command line `pydoc GAsympy` for the generation of a text file or `pydoc -w GAsympy` for the generation of a html file documenting GAsympy in the GAsympy directory.

## 8 GAcalc.py - A Calculator for Geometric Algebra

If one is not familiar with python programming a simple Geometric Algebra calculator is included with symbolicGA. To run `GAcalc.py` use the command line `GAcalc.py [savefile]`. `savefile` is an optional command line argument that allows one to save or restore their calculator session. If the file `savefile` does not exist `GAcalc` creates it and saves the current `GAcalc` session in it. If `savefile` exists, `GAcalc` reads it and executes the commands saved from a previous session restoring and making available all the quantities calculated in the previous session to the current session. Note one should never use a `savefile` name that ends in `.py` since this causes problems in the python interpreter that will cause the calculator to lock up. An example of a `GAcalc` session follows:

```
$ GAcalc.py session.sav
```

```

In: make_symbols('x0 x1 x2')
Out: [x0 x1 x2]
In: x = x0*a0+x1*a1+x2*a2
Out: x = {x0}a0+{x1}a1+{x2}a2
In: y = a0^a1^a2
Out: y = a0^a1^a2
In: x^y
Out: 0
In: y = a0^a1
Out: y = a0^a1
In: x^y
Out: {x2}a0^a1^a2
In: z = x+y
Out: z = {x0}a0+{x1}a1+{x2}a2+a0^a1
In: z = z+(a0^a1^a2)
Out: z = {x0}a0+{x1}a1+{x2}a2+a0^a1+a0^a1^a2
In: z.even()
Out: a0^a1
In: z.odd()
Out: {x0}a0+{x1}a1+{x2}a2+a0^a1^a2
In: z.rev()
Out: {x0}a0+{x1}a1+{x2}a2-a0^a1-a0^a1^a2
In: z = z+1
Out: z = 1+{x0}a0+{x1}a1+{x2}a2+a0^a1+a0^a1^a2
In: ?

```

The default basis for GAcac is 'a0 a1 a2' with the corresponding default metric. This can be overridden by inputting `MV.setup(your basis,your metric)`. Also it is absolutely critical to enclose operations with `^` and `|` in parenthesis since they have a lower priority than `+`, `-`, and `*`.