$\mathcal{GA}$lgebra Primer

Alan Macdonald
Emeritus Professor of Mathematics
Luther College
Decorah, Iowa USA
mailto:macdonal@luther.edu
faculty.luther.edu/~macdonal

January 24, 2018

Abstract

This document describes the installation and basic use of the geometric algebra/calculus Python module *$\mathcal{GA}$lgebra* written by Alan Bromborsky. It was written to accompany my texts *Linear and Geometric Algebra* and *Vector and Geometric Calculus*.

This is only an introduction to the module; many features are not covered. In some situations there are simpler approaches to those described here. But to include them would complicate this introduction. For complete documentation see GAlgebra.pdf, which is distributed with $\mathcal{GA}$lgebra.

New features may well be added to $\mathcal{GA}$lgebra. There is even a likely syntax change in the offing. So please check back for new versions of this document.

I encourage feedback and will post updated versions of this document as appropriate.

Contents

SymPy is a computer algebra system written in the popular computer programming language Python. It provides *symbolic* computation capabilities. For example, it can invert symbolic matrices. $\mathcal{GA}$lgebra adds symbolic geometric algebra and calculus capabilities to SymPy.

This primer does not teach Python or SymPy programming. It describes only what is necessary to accomplish simple linear algebra, geometric algebra, vector calculus, and geometric calculus computations with $\mathcal{GA}$lgebra.

1 Installation

You will need Python, SymPy, and $\mathcal{GA}$lgebra. Jupyter (formerly IPython) is optional. All are free, multiplatform, and downloadable. You will also need a program editor unless you plan to use Jupyter exclusively – see below.

Python. Install the latest Python 2.7 version (Python 3 will not work) from `https://www.python.org/downloads/`.

The Doc folder of a Python installation contains Python documentation. Online documentation is at `https://docs.python.org/2/download.html`.

A useful Python tutorial: `http://www.tutorialspoint.com/python/index.htm`.

SymPy. To install Sympy, open a command line in your Python27\Lib\Scripts folder and run `pip install sympy`. To update: `pip install sympy --upgrade`.
SymPy capabilities: `https://en.wikipedia.org/wiki/SymPy`.
Full documentation: `http://docs.sympy.org`.
Tutorial: `https://asmeurer.github.io/scipy-2014-tutorial/html/index.html`.
`mpmath` enables floating-point arithmetic with arbitrary precision in SymPy.

Install: `pip install mpmath`. Documentation: `http://mpmath.org/`.

$\mathcal{GA}$lgebra. At `https://github.com/brombo/galgebra` pull down the "Clone or download" menu and choose "Download ZIP". Copy the folder galgebra-master to the Python subfolder Lib\site-packages.

Open a command prompt in the galgebra subfolder of galgebra-master. On a Windows system run "python setgapth.py". On Linux and OS X systems enter and run the command "sudo python setgapth.py".

Program Editor. Geany is one possibility. It is cross-platform. Download and install it from `http://www.geany.org/`. There are 32- and 64-bit versions.

For printing to a console (see Section 4) Geany must know the location of the console program and configure it. This happens automatically on Linux and OS X, but not Windows. For Windows download and install ConEmu (`http://conemu.github.io/`). In Geany go to Edit/Preferences/Tools/Terminal and enter the full path (your choice) of conemu's exe file (in quotes), followed by "/WndW 180 /cmd %c" (no quotes).

Jupyter (formerly IPython) Notebook. The Jupyter Notebook provides a way to do Python programming *interactively*. It runs in a web browser. See Section 5.

To install Jupyter run `pip install "ipython[notebook]"` in a command prompt.

Notation. This document will use lower case italic for scalars (e.g., s), lower case bold for vectors (e.g., $\mathbf{v}$), upper case bold for blades (e.g., $\mathbf{B}$), and upper case italic for general multivectors (e.g., M). Python statements will appear in `this font`.

2 Algebra

2.1 Linear Algebra

Type the Python program below into your editor. The program defines the matrix $M = \left[\begin{smallmatrix} 1 & m \\ 3 & 4 \end{smallmatrix}\right]$ and then prints M^{-1}. The first line gives the program access to SymPy.

```
from sympy import *
m = symbols('m', real=True)  # Anything following a # is a comment
     # Without real=True, symbols are complex numbers.
M = Matrix( [ [1,m],[3,4] ] )  # Extra spaces inserted for clarity
print M.inv()
```

If you use Geany, press F5 in to run the program. You will be prompted to give the program a name. (Use a "py" extension.) The output is
[1+3m/(4-3m), -m/(4-3m)]
[-3/(4-3m), 1/(4-3m)]
Thus

$$M^{-1} = \begin{bmatrix} 1 + \frac{3m}{4-3m} & \frac{-m}{4-3m} \\ \frac{-3}{4-3m} & \frac{1}{4-3m} \end{bmatrix} = \frac{1}{4-3m} \begin{bmatrix} 4 & -m \\ -3 & 1 \end{bmatrix}.$$

We have used the *symbol* `m` in M. Symbols must be *declared.* One way to do this is with a `symbols` statement, as above. You can declare several symbols at once, e.g., `x1,x2,m,z = symbols('x1 x2 m z', real=True)`

Elementary matrix methods. SymPy provides several:

```
M + N      # sum
M * N      # product
M.inv()    # inverse
M.T        # transpose
M.det()    # determinant
M.rank()   # rank
```

Vector methods: norm, inner product. Implement vectors as matrices:

```
u = Matrix([1,2,3])   # A vector
v = Matrix([4,5,6])   # A vector
```

`print u.norm().evalf(3)`
Output: 3.74
`print u.dot(v)`
Output: 32
`print u.cross(v) # 3D only`
Output: Matrix([-3], [6], [-3])

Span. The `rref` method computes a basis for the span of the row vectors of a matrix. ("`rref`" is an abbreviation for *reduced row echelon form*.)

```
A = Matrix([ [1,2,-1], [-2,1,1], [0,5,-1] ])
print A.rref()[0]
```

Output (condensed): ([1, 0, -3/5] [0, 1, -1/5] [0, 0, 0])

The vectors `[1, 0, -3/5]` and `[0, 1, -1/5]` form a basis for the two dimensional span of the three row vectors of A.

Least squares.

```
from mpmath import lu_solve, matrix
A = matrix([[ 0, 1], [ 1, 1], [ 2, 1], [3, 1]])
b = matrix([[-1], [0.2], [0.9], [2.1]])
print lu_solve(A, b))
```

Output: [1.0] [-0.95] (Least squares line: $y = 1x - 0.95$)

Characteristic polynomials.

```
x = symbols('x')
M = Matrix([ [1,2], [2,1] ])
charpoly = (x*eye(2) - M).det() # eye(2) = 2 x 2 identity
print charpoly
```

Output: x**2 - 2*x - 3

```
print factor(charpoly)
```

Output: (x - 3)*(x + 1)

Singular value decomposition.

```
from mpmath import *
mp.dps = 4    # Set precision
A = matrix([[2, -2, -1], [3, 4, -2], [-2, -2, 0]])
U, S, V = svd_r(A).   # _r for real matrix; _c for complex
```

Simplify trigonometric expressions. Use the function `trigsimp` (which is not perfect). For example,

```
x = symbols('x')
print trigsimp(sin(x)**2 + cos(x)**2)
```

Output: 1

Functions. Here is a simple Python program using a function:

```
def absolute_value(n):
  if n < 0:
    n = -n
  return(n)
n = -1
print absolute_value(n), n  # Output: 1 -1
```

Execution starts with `n = -1`. The output shows that the value of `n` is unchanged by the function. Other object types can change. This has to do with the distinction between *mutable* and *immutable* objects in Python.

Indentations always follow colons. They are essential; leave one out and you will get a syntax error.

The following linear algebra functions need another import:

```
from mv import *
```

Systems of linear equations. `rref` (described above) also solves systems of linear equations. In this context the output from `rref` is not well formatted for human readers. The function `printrref` assumes that `rref`'s output is from a system of equations and prints it in a readable form.

As an example, consider the system $\begin{bmatrix} 1 & 2 & -1 & 2 \\ -2 & 1 & 1 & 0 \\ 2 & 0 & -2 & 4 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -1 \\ 1 \end{bmatrix}$. The *augmented matrix* of the system consists of the coefficient matrix augmented with the column vector on the right side. Assign it to `A` and `printrref` it:

```
A = Matrix([ [1,2,-1,2,4], [-2,1,1,0,-1], [2,0,-2,4,1] ])
printrref(A, 'wxyz')
```

Output:
```
1w + 0x + 0y + -2z = 9/4
0w + 1x + 0y +  0z = 7/4
0w + 0x + 1y + -4z = 7/4
```

The output is another system of equations. This system has two important properties. First, it has the same solutions as the original. Second, the solutions can be read directly from its equations. Starting from the first equation of our example, $w = 2z + 9/4$, $x = 7/4$, $y = 4z + 7/4$, with z not further constrained. Set it equal to t. Then the solution is shown at the right.

$$\begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 4 \\ 1 \end{bmatrix} t + \begin{bmatrix} 9/4 \\ 7/4 \\ 7/4 \\ 0 \end{bmatrix}$$

Eigenvalues and eigenvectors. SymPy provides `M.eigenvects()` for the eigenvectors of matrix M. But its output is not well formatted for human reading. The statement `printeigen(M)` will print the eigenvalues of a matrix M, their multiplicities, and their eigenvectors.

Gram-Schmidt orthogonalization. It is applied to a list of vectors, each implemented as a matrix:

```
L = [Matrix([1,2]), Matrix([3,4])]
print GramSchmidt(L)
```

Output: [[1] [2], [4/5] [-2/5]]

A second argument set to `True` will normalize the eigenvectors:

```
print GramSchmidt(L, True)
```

Output: $[[\sqrt{5}/5]\ [2 * \sqrt{5}/5], [2 * \sqrt{5}/5]\ [-\sqrt{5}/5]]$

This output is not well formatted for human readers. `printGS` will print the output of GramSchmidt in decimal form:

```
printGS(L, True)   (Note change from earlier version)
```

Output: [[0.447, 0.894] [0.894, -0.447]]

```
u = Matrix([1,2,3])   # A vector
v = Matrix([4,5,6])   # A vector
print correlation(u,v)
```

Output: 1

2.2 Geometric Algebra

To use the geometric algebra facilities of $\mathcal{GA}$lgebra, first create a specific geometric algebra. This code creates the standard 3D geometric algebra and names it g3:

```
from sympy import *
from ga import Ga  # import galgebra
g3coords = (x,y,z) = symbols('x y z')
g3 = Ga('ex ey ez', g=[1,1,1], coords=g3coords)  # Create g3
     # [1,1,1]: norms squared of basis vectors (assumed orthogonal)
     # Example of 2D nondiagonal metric: g = '0 1, 1 0'
(ex, ey, ez) = g3.mv()
```

The two sets of coordinate names in the program above, (x y z) and 'x y z', are the same. The same is true of the basis vector names, 'ex ey ez' and (ex ey ez). See Section 4, Printing, for reasons to make them different.

The program produces no output. Add these lines:

```
A = y*ex + 3*ex*ey
B = x*ey
print A*B
```

Output: 3*x*ex + x*y*ex^ey.

Substitute. Sometimes you want to substitute specific values for variables. Example: `print (A*B).subs({x:1,y:2})` produces 3*ex + 2*ex^ey. Note that `subs` leaves variables unchanged, e.g., `A` is unchanged after `print A.subs({y:2})`.

Arithmetic Operators. If you see the arithmetic expression $2 + 3 \times 4$ you know to multiply 3×4 first and then add 2. This is because mathematics has a *convention* that multiplication comes before addition; multiplication has higher *precedence* than addition. If you want to add first, write $(2 + 3) \times 4$.

The table shows $\mathcal{GA}$lgebra's geometric algebra arithmetic operators.[1] They are given in precedence order (imposed by Python), high to low.[2] Plus and minus are grouped because they have the same precedence, as do $<$ and $>$.

$*$	geometric product
$+ -$	add, subtract
$\wedge$	outer product
$\|$	inner product
$<$	left contraction

The high precedence of $+ -$ causes a problem. Consider the simple expression $\mathbf{u} + \mathbf{v} \cdot \mathbf{w}$. $\mathcal{GA}$lgebra evaluates it as $(\mathbf{u} + \mathbf{v}) \cdot \mathbf{w}$. If you intend $\mathbf{u} + (\mathbf{v} \cdot \mathbf{w})$, as you probably do, then you must use the parentheses. As another example, $\mathcal{GA}$lgebra evaluates $\mathbf{u} \cdot \mathbf{v} * \mathbf{w}$ as $\mathbf{u} \cdot (\mathbf{v} * \mathbf{w})$. If you intend $(\mathbf{u} \cdot \mathbf{v}) * \mathbf{w}$, then you must use the parentheses. (For many authors $\mathbf{u} \cdot \mathbf{v} * \mathbf{w}$ *does* mean $(\mathbf{u} \cdot \mathbf{v}) * \mathbf{w}$.)

As a general rule, you must put parentheses around terms with inner or outer products, to "protect" them from the high precedence $\pm$'s bounding them, as in the $\mathbf{u} + \mathbf{v} \cdot \mathbf{w}$ example. And remember that within terms the geometric product has higher precedence than the inner and outer products, as in the $\mathbf{u} \cdot \mathbf{v} * \mathbf{w}$ example.

Algebras.py. This file is distributed with $\mathcal{GA}$lgebra. It contains code to create many different geometric algebras, including g3 and all others used in this document, as well as others not covered here: homogeneous, spacetime, and conformal algebras.

[1] Use "$<$" for the product called the inner product and denoted "$\cdot$" in my books and this document. Use "$|$" for the product called the inner product and denoted "$\cdot$" in, e.g., books by Hestenes and Sobczyk and by Doran and Lasenby.

[2] Click for a complete list of Python's operator precedences.

Multivector Functions

To use (most of) these, first `from mv import *`. There is an alternate form for most, which do not require the import. Examples: `M.dual()` and `M.grade(r)`.

In the list below, `M` is a multivector; `A, B` are blades; `v1, v2` are vectors; and `ga` is a geometric algebra, e.g., `g3`.

`ga.E()`	Outer product of the basis vectors of `ga`.
`ga.I()`	Unit pseudoscalar of `ga`, i.e., `ga.E()` normalized.
`dual(M)`	M^*. Returns MI. My books use $M^* = MI^{-1}$.
	For this, issue `Ga.dual_mode ('Iinv+')`after imports.
`even(M)`	Even grades of M
`exp(M)`	e^M. M^2 must be a scalar constant.
	If $M^2 > 0$, use `exp(M,'+')`
`grade(M,r)`	$\langle M \rangle_r$
`grade(M)`	$\langle M \rangle_0$
`inv(M)`	M^{-1}
`norm(M)`	$\|M\|$
`norm2(M)`	$\|M\|^2$
`odd(M)`	Odd grades of M
`proj(B,A)`	$\mathsf{P}_{\mathbf{B}}(\mathbf{A})$. Projection of $\mathbf{A}$ on $\mathbf{B}$.
`rot(itheta,A)`	$\mathsf{R}_{i\theta}(\mathbf{A})$. Rotation of $\mathbf{A}$ by angle $i\theta$.
`refl(B,A)`	$\mathsf{F}_{\mathbf{B}}(\mathbf{A})$. Reflection of $\mathbf{A}$ in $\mathbf{B}$.
`scalar(M)`	$\mathcal{G}\mathcal{A}$lgebra scalar $\rightarrow$ sympy scalar.
`com(A,B)`	Commutator: $[A, B] = AB - BA$
`cross(v1,v2)`	Cross product
`Nga(M, prec=k)`	Round decimals in M to k significant figures.
`rev(M)`	$M^\dagger$. Reverse of M.
`ReciprocalFrame(basis)`	basis is a list of vectors enclosed in parentheses.
`ga.r_basis`	List of reciprocal basis vectors of `ga` basis.
	Each is expanded in the `ga` basis.

There are also member function versions of these functions, e.g., `M.even()`.

Linear transformations. The following three examples create a linear transformation (outermorphism) `L` on the geometric algebra `g2`. The matrix of the transformation with respect to the basis {ex, ey} is also shown.

`L = g2.lt('A')`. Matrix: $\begin{bmatrix} A_{xx} & A_{xy} \\ A_{yx} & A_{yy} \end{bmatrix}$ (because `g2` has coordinates x and y).

An optional second parameter `mode = 's'` or `mode = 'a'` will generate a general symmetric or antisymmetric (skew) matrix, respectively.

`L = g2.lt([[0,2],[1,1]])`. Matrix: $\left[\begin{smallmatrix} 0 & 1 \\ 2 & 1 \end{smallmatrix}\right]$.

`L = g2.lt([2*ey,ex+ey])`. Matrix: $\left[\begin{smallmatrix} 0 & 1 \\ 2 & 1 \end{smallmatrix}\right]$.

An optional second parameter `f=True` to `lt` makes the linear transformation a function of the coordinates.

If M is a multivector (not necessarily a vector), then $\mathsf{L}(M)$ is the result of the outermorphism `L` applied to M.

Linear transformations can be added (+), subtracted (−), and composed (*). `L.det()` (determinant), `L.adj()` (adjoint), `L.tr()` (trace), and `L.matrix()` are also available.

General Multivectors

$\mathcal{GA}$lgebra allows assignments of variables to *specific* multivectors as in `A = y*ex + 3*ex*ey` above. $\mathcal{GA}$lgebra can also create *general* multivectors. For example, this code creates and prints a general vector Python variable V:

```
V = g2.mv('V', 'vector')
print V
```

Output: V__x*e_x + V__y*e_y
The double underscore __ is explained in Section 4.

The first parameter is a string, the printing name of the variable. An optional third parameter `f=True` makes the coefficients a function of the coordinates, i.e, makes the multivector a function of the coordinates. Here are the options for the second parameter:

2nd	Result
'scalar'	scalar
'vector'	vector
'bivector'	bivector
n	grade n multivector
'pseudo'	pseudoscalar
'even'	even multivector (spinor)
'mv'	general multivector

The scalar result in the top row is a general scalar multivector, a member of the geometric algebra. This is different from a SymPy scalar.

In addition, `g2.mv(c)`, where `c` is a specific scalar, is available.

General multivectors can be useful to test a conjecture about geometric algebra, especially when first learning. For example, let $\mathbf{B}$ be a bivector. After finding that $\mathbf{v}\cdot\mathbf{B}$ is in $\mathbf{B}$ for several vectors $\mathbf{v}$, one might wonder if this is always so. The following code proves this.

```
v = g3.mv('v', 'vector')   # Construct a symbolic vector in g3.
B = g3.mv('B', 'bivector') # Construct a symbolic bivector in g3.
W = (v < B) ∧ B            # W = 0 ⇔ v · B ∈ B.
print W.simplify()         # Do not use simplify(W).
```

Output: 0.

Thus the statement "$\mathbf{v}\cdot\mathbf{B}$ is in $\mathbf{B}$" is true. The code is for $\mathbb{G}^3$. Does the statement remain true in higher dimensions? Not for general bivectors $\mathbf{B}$. But recall that a general bivector $\mathbf{B}$ in $\mathbb{G}^3$ is an outer product of two vectors. So the statement in $\mathbb{G}^3$ is about three vectors. Thus it remains true in higher dimensions if $\mathbf{B}$ is an outer product of two vectors. For then it involves three vectors in some 3D subspace.

Try adding code to show that $\mathbf{v}\cdot\mathbf{B}$ is orthogonal to $\mathbf{v}$.

3 Calculus

3.1 Vector Calculus

Differentiation, including partial differentiation.

```
x,y = symbols('x y') # Define the symbols you want to use.
print diff(y*x**2, x)
```

Output: 2*x*y

```
print diff(diff(y*x**2,x),y)
```

Output: 2*x

```
f = y*x**2
print diff(f,x)
```

Output: 2*x*y

Jacobian. Let X be an $m \times 1$ matrix of m variables. Let Y be an $n \times 1$ matrix of functions of the m variables. These define a function $\mathbf{f}\colon X \in \mathbb{R}^m \to Y \in \mathbb{R}^n$. Then `Y.jacobian(X)` is the $n \times m$ matrix of $\mathbf{f}'_{\mathbf{x}}$, the differential of $\mathbf{f}$ at $\mathbf{x}$.

```
r, theta = symbols('r theta')
X = Matrix([r, theta])
Y = Matrix([r*cos(theta), r*sin(theta)])
print Y.jacobian(X)          # Print 2 × 2 Jacobian matrix.
print Y.jacobian(X).det()  # Print Jacobian determinant (only if m = n).
```

Sometimes you want to differentiate `Y` only with respect to some of the variables in `X`. Then replace `X` in `Y.jacobian(X)` with only those variables. For example, `print Y.jacobian([r])` produces the 2×1 matrix $\left[\begin{smallmatrix}\cos\theta\\ \sin\theta\end{smallmatrix}\right]$.

Integration. `integrate(f, x)` returns an indefinite integral $\int f\,dx$.
`integrate(f, (x, a, b))` returns the definite integral $\int_a^b f\,dx$.

```
x = Symbol('x')
print integrate(x**2 + x + 1, x)
```

Output: x**3/3 + x**2/2 + x

Iterated integrals.
This code evaluates $\int_{x=0}^{1}\int_{y=0}^{1-x}(x+y)\,dy\,dx$:

```
x, y = symbols('x y')
I1 = integrate(x + y, (y, 0, 1-x))
I2 = integrate(I1, (x, 0, 1))
```

evalf.

```
print log(10), log(10).evalf(3)
```

Output: log(10) 2.30

3.2 Geometric Calculus

`grad = g3.grad` assigns to `grad` (your choice) the gradient operator of the geometric algebra `g3`, $\nabla = \mathbf{e}_x\partial_x + \mathbf{e}_y\partial_y + \mathbf{e}_z\partial_z$. If F is multivector valued, then `grad*F`, `grad<F`, and `grad^F` are defined and are called the gradient, divergence, and curl of F, respectively.

The directional derivative of F in the direction `a` is `(a<grad)*F`.

Curvilinear coordinates. Curvilinear coordinates are implemented by creating an appropriate geometric algebra. For example, this code creates sp3, the standard geometric algebra in $\mathbb{R}^3$, in spherical coordinates:

```
sp3coords = (r, phi, theta) = symbols('r phi theta')
sp3 = Ga('er ephi etheta', g=None, coords=sp3coords, \
  X=[r,r*sin(phi)*cos(theta),r*sin(phi)*sin(theta),r*cos(phi)],norm=True)
```

\" is Python's line continuation character.

"norm=True" returns normalized basis vectors, often denoted $\hat{\mathbf{r}}, \hat{\boldsymbol{\phi}}, \hat{\boldsymbol{\theta}}$.

Mathematics naming convention: ϕ colatitude, θ longitude.

```
(er, ephi, etheta) = sp3.mv()
sp3grad = sp3.grad
```

Now `sp3grad` $= \mathbf{e}_r\partial_r + \mathbf{e}_\phi r^{-1}\partial_\phi + \mathbf{e}_\theta(r\sin\phi)^{-1}\partial_\theta$, the gradient operator of `sp3`.

Here is a different `Ga` statement to create sp3:

```
sp3=Ga('er ephi etheta',g=[1,r**2,r**2*sin(theta)**2],sp3coords,norm=True)
```

Manifolds. This example creates the unit sphere sp2 in $\mathbb{R}^3$ as a submanifold of the geometric algebra g3 from Section 2.2:

```
sp2coords = (phi,theta) = symbols('phi theta', real=True)
sp2param = [sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi)]
  # Parameterize sp2 in terms of the x,y,z coordinates of g3
sp2 = g3.sm(sp2param, sp2coords, norm=True)
(ephi, etheta) = sp2.mv() # sp2 basis vectors
(rphi, rtheta) = sp2.mvr() # sp2 reciprocal basis vectors
sp2grad = sp2.grad
```

Now `sp2grad` $= \mathbf{e}_\phi\partial_\phi + \mathbf{e}_\theta(\sin\phi)^{-1}\partial_\theta$, the gradient operator of `sp2`. Note that it is the restriction of `sp3grad` to the sphere.

Multivectors can be expressed in either the sp2 basis (ephi, etheta) or the g3 basis (ex, ey, ez).

Here is another way to create the unit sphere in $\mathbb{R}^3$, this time as a submanifold of the geometric algebra sp3 from Section 2.2:

```
sp2coords = (p,t) = symbols('p t', real=True)   # p = phi, t = theta
sp2param = [1, p, t] # Parameterization of unit sphere
sp2 = sp3.sm(sp2param, sp2coords)
```

4 Printing

$\mathcal{GA}$lgebra has two modes of output: to a console (terminal) or to a pdf with beautiful LaTeX typesetting.

Subscripts and superscripts. With the code

```
sp2coords = (ph,th) = symbols('phi theta', real=True)
```

the short "th" is used in the program, e.g., `print th`. However, and this is the point, the print statement sends "theta" to a console, and "θ" to a pdf.

Similarly, with the code

```
sp3 = Ga('e_r e_phi e_theta', ...  )
(er, eph, eth) = sp3.mv()
```

the "eth" is used in the program, e.g., `print eth`. However, the print statement sends "e_theta" to a console and "$\mathbf{e}_\theta$" to a pdf.

The statement

`print g2.mv('V', 'vector')` sends "V__x*e_x + V__y*e_y" to the console and "$V^x\mathbf{e}_x + V^y\mathbf{e}_y$" to a pdf.

With console output, '\n ' (note space) in a string in a print statement starts a new line.

Fmt. The command `Fmt(n)` specifies how multivectors are split over lines:

$n = 1$: The multivector is printed on one line. (The default.)

$n = 2$: Each grade of the multivector is printed on a separate line.

$n = 3$: Each component of the multivector is printed on a separate line.

The $n = 2$ and $n = 3$ options are useful when a multivector will not fit on one line.

The code `print A.Fmt(n)` will print A as specified. And `print A.Fmt(n,'A')` will print the string A = followed by A, as specified. You can print a variable with one n and later with another. An earlier `Fmt(n)` remains in force after these print versions of `Fmt`.

Enhanced printing. With enhanced printing to the console, multivector bases, functions, and derivatives are printed in different colors for easier reading. To invoke enhanced printing, issue these commands:

```
from printer import Eprint
Eprint() # right after the import statements
```

The color coding is automatic on Linux and OS X. But on Windows the default colors of ConEmu (Section 1) should be changed. For this, right click on its Title Bar. Choose Settings, then Colors. I use Text: 0; Back: #7; Popup: #0; Back: #7; and ☑ Extend foreground colors with background #13. If you find a color combination that looks better to you, let me know.

LaTeX output. When a program using LaTeX mode ends, its output is opened in the default pdf reader. (In Windows, you must close the pdf reader, or at least the tab for your file, before rerunning your program. Otherwise your program will hang.) It is helpful, but not necessary, to know a bit of LaTeX for this. Of course you need a LaTeX system on your computer.[3]

You need these statements to use LaTeX printing:

```
from printer import *
Format() # after the import
xpdf() # last statement of program.
```

Here is an example using LaTeX with $\mathcal{GA}$lgebra. When printing a string, an underscore "_" designates a subscript. A caret "^" (not a double underscore) designates a superscript. The statement

```
print r'\alpha_1\bm{X}/\gamma_r^3'
```

produces the output $\alpha_1 \boldsymbol{X}/\gamma_r^3$. Note the r preceding the string. It prevents certain undesirable (from $\mathcal{GA}$lgebra's point of view) Python processing of strings with backslashes. An alternative is to everywhere replace a "\" with "\\".

If the string contains an "=", e.g., `r'XXX = YYY'`, then substitutions are made in `XXX` (only) according to the table. Thus

grad	∇
^	$\wedge$
*	
<	⌋

```
r'grad A^B | * = grad A^B | *'
```

prints as $\nabla A \wedge B \cdot = \text{grad} A^B | *$

A newline $\backslash n$ cannot appear in a string preceded by an r. Instead use `r'A' + \ n' + r'B'` to split `'AB'` into two lines. The +'s glue (concatenate) strings.

The parameters `Format(Fmode=True, Dmode=True)` give additional formatting options. Use just one or both.

`Fmode=True`. Use f, not the default $f(x,y)$.

`Dmode=True`. Use $\partial_x f$, not the default $\frac{\partial f}{\partial x}$.

The default output page size is $8\frac{1}{2} \times 11$ (North American letter). You can change this. For example, `xpdf(paper=(8.5,100))` prevents page breaks in a pdf file to be read in a reader.

The file Symbols.pdf lists common LaTeX symbols. It is distributed with $\mathcal{GA}$lgebra.

[3] TeX Live is known to work, as are MiKTeX on Windows and MacTex on OS X. I think that it is only necessary that the LaTeX system provide pdflatex. Please let me know if you find otherwise.

5 Jupyter Notebook

IPython has been renamed Jupyter. This section is not an Jupyter Notebook tutorial. (For one thing, I don't know enough to write one.) But it will help get you get started with the software. Please send me suggestions about ways to make it easier to use the notebook on Windows, Linux, and/or OS X.

Description and many links: `https://en.wikipedia.org/wiki/IPython`
Introduction: `http://ipython.org/notebook.html`
Quick tutorials:
`http://ipython.org/ipython-doc/1/interactive/notebook.html`
Another Click (The URL is too long.)
Documentation:
`http://ipython.org/documentation.html`.
`http://ipython.org/ipython-doc/stable/notebook/index.html`.
Another Click.
Example presentations: `http://ipython.org/presentation.html`.

Start Notebook. Jupyter files have an ipynb extension. After installation of Jupyter, jupyter-notebook.exe is in the Python27/Scripts folder. On my PC, I associate ipynb files to it. Then I can double click on an ipynb file to open Jupyter. To start a new notebook I open jupyter-notebook.exe.

Enter Statements. Press "enter" to start a new line.

Execute. To execute a notebook cell, press "shift+enter"

Output. All output is typeset by LaTeX. Thus, as described in the LaTeX page in Section 4, you need to execute these lines:

```
from printer import *
Format() # after all imports
```

Do not use `print`: Use `ex`, not `print ex`; use `A.Fmt()`, not `print A.Fmt()`. Do not use `xpdf()`.

If you want to produce output more than once in a cell (e.g., in a loop) use, e.g., `display(ex)`. You must first `from IPython.display import display`.

If ph is a variable with value $\pi/4$, then `Latex(r'$\phi = ' + str(ph) + '$')` produces the output $\phi = \frac{\pi}{4}$.

I have written several "get started" notebook files, one for each geometric algebra in Algebras.py (See Section 2.2). For example, g3.ipynb imports SymPy and $\mathcal{GA}$lgebra and sets up the geometric algebra g3. I can open the notebook and execute g3's cell, and be ready to make g3 calculations interactively. Better, make g3.ipynb read-only. Then when starting a new g3 notebook, copy g3.ipynb to MyNewNotebook.ipynb (not read-only) and use it for the notebook.

$\mathcal{GA}$lgebra's option to print to a pdf is not available in a notebook. However, it is possible to convert a Notebook to a Python program (and other formats):
`http://ipython.org/ipython-doc/1/interactive/nbconvert.html`.