$\mathcal{GA}$lgebra Primer

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

August 12, 2020

Abstract

This document describes the installation and basic use of the geometric algebra/calculus Python package *$\mathcal{GA}$lgebra*. It was written to accompany my texts *Linear and Geometric Algebra* and *Vector and Geometric Calculus*. Some, but not much, acquaintance with programming is helpful in learning to use $\mathcal{GA}$lgebra.

This is only an introduction to $\mathcal{GA}$lgebra; many features are not covered. In some situations there are simpler approaches to those described here. But to include them would complicate this introduction.

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

$\mathcal{GA}$lgebra was originally written by Alan Bromborsky. It is now under development by the *Pythonic Geometric Algebra Enthusiasts*, PyGAE. (Active links – external and internal – appear in blue.)

This is the first version of the primer devoted to the PyGAE $\mathcal{GA}$lgebra. So there are bound to be problems. Do not hesitate to contact me with questions or problems. I thank Gregory Grunberg for help with the transition.

The Python module SymPy (Symbolic Python) is a computer algebra system written in the popular computer programming language Python. It provides *symbolic* computation capabilities, similar to Mathematica and Maple. For example, it can invert symbolic matrices. (See p. 7.)

$\mathcal{GA}$lgebra adds symbolic geometric algebra and calculus capabilities to SymPy. It produces beautifully formatted mathematical output using LaTeX, processed by MathJax.

My intent is that to do the exercises and problems in my books you need not know more programming than is documented here.

Contents

1 Install

You need Python ≥3.6, SymPy, $\mathcal{GA}$lgebra, and Jupyter Lab.[1,2]

• Install Python. Download and execute the Python installation file. Documentation. Tutorial. I am not recommending that you study these.

The Python distribution includes the package manager *pip*. Use it below to install SymPy, Jupyter Lab, and $\mathcal{GA}$lgebra. Run pip from a command line in your Python3/Scripts folder. To show pip's options, type "pip".

• Install SymPy: `pip install sympy`. Upgrade: `pip install sympy --upgrade`. SymPy capabilities. Full documentation. SymPy tutorial.

• Install $\mathcal{GA}$lgebra: `pip install galgebra`.
Documentation. It is incomplete and not always accurate.

• Install Jupyter Lab: `pip install jupyterlab`.
Jupyter Lab home, Getting started, Documentation, Wikipedia.

I often find Google more help than the documentation links above. For example, I Googled "latex small matrix" when I wanted to know how to make the small matrix on p. 5.

Jupyter Lab is a *notebook environment*: it runs in a web browser and provides *interactive* Python programming capability. The notebook files have an ipynb extension. I keep my notebook files in one folder, the "*root*", and its subfolders. Jupyter Lab allows navigation only in the subtree of the root – something about security.

Download GAfiles.zip. Unzip it into your root folder. One of the files in it is gprint.py. Move it to the `/Lib/site-packages/galgebra` subfolder of your Python distribution, where ga.py is.

Your installation is complete.

Here is one way to proceed on a PC. Create a shortcut to jupyter-lab in the Scripts folder of your Python installation. Right click on the shortcut, select "properties", and enter the location of your root in "Start in". Click on the shortcut and Jupyter will open, displaying your root in the left sidebar. If you don't see them, use the "View menu".

[1] It is possible to use a program editor instead of Jupyter Lab. $\mathcal{GA}$lgebra's documentation for this is here. You will need to install a LaTeX distribution. I do not recommend an editor when $\mathcal{GA}$lgebra is used with my books.

[2] There are many distributions of Python with extra bells and whistles. Anaconda is large (300 GB), free, multiplatform, and popular. Anaconda includes SymPy and Jupyter Lab. $\mathcal{GA}$lgebra must be installed separately.

2 Jupyter Lab

This section will get you get started with Jupyter Lab. You will save time later if you spend a little time with this page now. Please send me suggestions about ways to make it easier to use Jupyter Lab.

$\mathcal{GA}$lgebra's output function is *gprint*. I have written several notebook files, each defining a specific geometric algebra. They are in your root folder. For example, g3.ipynb defines the geometric algebra $\mathbb{G}^3$, and names it g3. I will refer to the files by name in this primer.

Jupyter Lab's help menu includes help for Python, SymPy, and Jupyter Lab.

New Line in Cell. Press "enter". **Execute a cell.** Press "shift+enter".

Keyboard Shortcuts. *Very* useful. To use one, *select* a cell by clicking to its left on the main window (avoid the vertical bar). Here are several:

X	Cut selected cells	DD	Delete selected cells
C	Copy selected cells	V	Paste cells below
Shift ↑	Continue selection above	Shift ↓	Continue selection below
A	Insert empty cell above	B	Insert empty cell below
Shift M	Merge selected cells	Ctrl S	Save notebook
Ctrl Z	Undo cell operation	Shift Ctrl Z	Redo cell operation

To open a list of commands, click at the left on the magnifying glass over a list.

Copy and paste. Select a cell or contiguous cells. Then copy and paste – even between different notebooks – using shortcuts or the Edit menu. Copy and paste text *in* cells the usual way, according to your operating system.

View folder. Click on the folder icon at the left side of the screen to view the folder from which your notebook was opened.

Export Notebook. In the *File* menu, choose *Export Notebook As...*. HTML works for me. PDF does not. (But "print to PDF" from a browser does.)

Collapse Cell. To the left of the active cell there is a vertical blue bar. If the cell has output, there is a second bar. Click on a bar to collapse a cell or its output. Click on the bar or ••• to expand. Several collapse/expand commands in the palette apply to all or all selected cells.

Checkpoints. Ctrl-S saves your notebook and makes a *checkpoint*. Later you can "Revert Notebook to Checkpoint" from the File menu.

3 Output

Skip this section for now and turn to Linear Algebra, Section 4.1. Return here when you are ready to use Jupyter Lab for Geometric Algebra, Section 4.2.

We will illustrate gprint's capabilities with GAlgebraOutput.ipynb in your root folder. Open it in Jupyter.

Execute Cell 1. Ignore comments if you like. The cell has no output. Don't proceed until it has finished, indicated by a vertical bar to the left of Cell 2.

Execute Cell 2. The cell defines the geometric algebra $\mathbb{G}^3$. Again no output. Use this cell as a template for defining other geometric algebras. Let's look at it:

`g3coords = (x,y,z) = symbols('x y z', real=True)` .

Define the coordinate names as symbols.

Without real=True, the coordinates will be complex numbers.

`g3 = Ga('\mathbf{e}', g=[1,1,1], coords=g3coords).`

This defines the geometric algebra $\mathbb{G}^3$ and names it g3.

Output basis vectors are bold **e**'s, according to the convention in my books.

g = [1,1,1]: norms squared of basis vectors (assumed orthogonal).

Optional parameter `wedge=False` displays the basis vector $\mathbf{e}_x \wedge \mathbf{e}_y$ as $\mathbf{e}_{xy}$, etc.

You might find this more readable, especially in higher dimensions.

Example of 2D nondiagonal inner products: g = '0 1, 1 0' for $\mathbf{e}_i \cdot \mathbf{e}_j = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$.

`(ex, ey, ez) = g3.mv()` Program names of basis vectors.

The two sets of coordinate names in the program above, (x y z) and 'x, y, z', are the same. We will see reasons to make them different later.

Execute Cell 3. `gprint` has two input modes: *string* and *$\mathcal{GA}$lgebra*. String input is enclosed in quotes – single or double. An "r" preceding a string prevents certain undesirable (from $\mathcal{GA}$lgebra's point of view) Python processing of strings with backslashes. LaTeX symbols, e.g., \cdot, are evaluated. There are two in the cell: r'\cdot' and '='.

$\mathcal{GA}$lgebra input is a single mathematics expression to be evaluated by $\mathcal{GA}$lgebra. There are three in the cell: ex, ey, and ex<ey. (A cell with only one output line, a single $\mathcal{GA}$lgebra expression, will produce output without a gprint.)

The very useful file Symbols.pdf lists common LaTeX symbols. There is a detailed LaTeX Wiki. Draw a symbol with your mouse and *Detexify* will try to determine its LaTeX symbol.

The left column of the table gives $\mathcal{GA}$lgebra's output symbols for some geometric algebra operators used in my books. The middle column gives string mode input for the output. The right column gives the $\mathcal{GA}$lgebra operator for the output. See Section 5.2 for grad.

$\cdot$	\cdot	<
$\wedge$	\wedge	^
		*
∇	\nabla	grad

Use the $\mathcal{GA}$lgebra operator "<" for the inner product "$\cdot$" of my books.

Execute Cell 4. The two input strings are the same, except that the second uses \text. Inspect the input and output and compare the output lines.

Execute the remaining cells to learn about input and output. Experiment!

The first cell. The first cell of all notebooks from GAfiles.zip are identical. They get all the pieces of $\mathcal{GA}$lgebra ready for use. Start every program with this cell. The second cell of each notebook defines a geometric algebra.

Here is another way to execute the first cell. The file GAlgebraInit.py from GAfiles.zip contains the cell. When starting a new program, execute `from GAlgebraInit import *` in the first cell. You will not see anything happening, but the cell will be executed.

If the cell needs to be changed, as is posible, you need not change all your programs, but only GAlgebraInit.py.

Fmt. This specifies how the terms of a multivector M are split across lines when output:

$n = 1$: All of M is output on one line.

$n = 2$: Each grade of M is output on a separate line.

$n = 3$: Each term of M is output on a separate line.

gFormat. Use one or both parameters for functions.

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

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

Also, gFormat defines several commands, listed below. DOESN'T WORK.

LaTeX macros defined in gprinter.py

```
\DeclareMathOperator{\Tr}{Tr}
\DeclareMathOperator{\Adj}{Adj}
\newcommand{\bfrac}[2]{\displaystyle\frac{#1}{#2}}
\newcommand{\lp}{\left (}
\newcommand{\rp}{\right )}
\newcommand{\paren}[1]{\lp {#1} \rp}
\newcommand{\half}{\frac{1}{2}}
\newcommand{\llt}{\left <}
\newcommand{\rgt}{\right >}
\newcommand{\abs}[1]{\left |{#1}\right | }
\newcommand{\pdiff}[2]{\bfrac{\partial {#1}}{\partial {#2}}}
\newcommand{\npdiff}[3]{\bfrac{\partial^{#3} {#1}}{\partial {#2}^{#3}}}
\newcommand{\lbrc}{\left \{}
\newcommand{\rbrc}{\right \}}
\newcommand{\W}{\wedge}
\newcommand{\prm}[1]{{#1}'}
\newcommand{\ddt}[1]{\bfrac{d{#1}}{dt}}
\newcommand{\R}{\dagger}
\newcommand{\deriv}[3]{\bfrac{d^{#3}#1}{d{#2}^{#3}}}
\newcommand{\grade}[1]{\left < {#1} \right >}
\newcommand{\f}[2]{{#1}\lp {#2} \rp}
\newcommand{\eval}[2]{\left . {#1} \right |_{#2}}$
```

The list might change. The official list is here. (Scroll down for it.)

Notation. This document uses 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 appear in `this font`.

In the examples below I can mostly cut from this pdf and paste into a Jupyter Lab cell. An exception is that I get an extra space after a single quote " ' ". Another is that " * " sometimes does not paste correctly.

4 Algebra

4.1 Linear Algebra

This Section 4.1 discusses SymPy and linear algebra. Section 4.2 will take up $\mathcal{GA}$lgebra and geometric algebra.

Open Jupyter Lab. Start a new notebook: pull down the File menu and select New Notebook. I named my file LinAlg.ipynb.

Type the following lines in the first cell, pressing Enter after each to advance to the next line. Or copy and paste them. Anything after a "#" is a comment.

```
from sympy import *  # Make SymPy available to the program.
from galgebra.mv import * # Explained later.
```

Execute the cell by pressing Shift+Enter. The cell produces no output.

Now type the Python statements below into the second cell and execute it. The program defines the matrix $M = \left[\begin{smallmatrix} 1 & m \\ 3 & 4 \end{smallmatrix}\right]$ and then outputs M^{-1}.

```
m = symbols('m', real=True)  # Without real=True, m is complex
M = Matrix( [ [1,m],[3,4] ] )  # Extra spaces inserted for clarity
M.inv()
```

Output: $\begin{bmatrix} \frac{4}{4-3m} & -\frac{m}{4-3m} \\ -\frac{3}{4-3m} & \frac{1}{4-3m} \end{bmatrix}$

The `m` in M is a *symbol*. 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      # difference
M * N     # product                | M**n       # integer power
M.inv()   # inverse                | M.T        # transpose
M.det()   # determinant            | M.rank()   # rank
c*M       # scalar multiplication  |
```

You might just scan the rest of this section to see what is possible. You can look up particular topics when you need them.

Several functions to follow are part of the `mv` module of $\mathcal{G}$*Algebra*. Thus we must import mv: `from galgebra.mv import *` .

Span. The `rref` method computes a basis for the span of the row vectors of a matrix. ("`rref`" is an abbreviation for **r**educed **r**ow **e**chelon **f**orm.)

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

Output: $\begin{bmatrix} 1 & 0 & -\frac{3}{5} \\ 0 & 1 & -\frac{1}{5} \\ 0 & 0 & 0 \end{bmatrix}$.

The nonzero rows form a basis for the span of the row vectors of A.

Least squares.

```
A = Matrix([[0, 1], [1, 1], [2, 1], [3, 1]])
b = Matrix([[-1], [0.2], [0.9], [2.1]])
A.LDLsolve(b)
```

Output: $\begin{bmatrix} 1.0 \\ -0.95 \end{bmatrix}$ (Least squares line: $y = 1.0x - 0.95$)

Characteristic polynomials.

```
x = symbols('x', real=True)
M = Matrix([ [1,2], [2,1] ])
cp = det(x*eye(2) - M) # cp = x^2 - 2x - 3. eye(2) = 2 x 2 identity matrix
factor(cp)  Output: (x - 3)(x + 1).
```

Simplify trigonometric expressions.

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

```
x = symbols('x', real=True)
trigsimp(sin(x)**2 + cos(x)**2)   Output: 1
```

Singular value decomposition.

```
from mpmath import svd
A = Matrix([[2, -2, -1], [3, 4, -2], [-2, -2, 0]])
U, S, V = svd(A)  % U, S, V all assigned values.
U  % The very small numbers in U are 0.
```

Functions. This is a simple Python function:

```
def absolute_value(n):
  if n < 0:
    n = -n
  return(n)
```

Colons in a program are always followed by indentations. They are essential: leave one out and you will get a syntax error. This executes the function:

```
m = -1
absolute_value(m), m  # Output 2 items   Output: (1 -1).
```

The output shows that `m` is *immutable*: its value remains `-1`, despite the function. Other objects are *mutable*.

Systems of linear equations. *rref* (described above) also solves systems of linear equations. 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 `rref` it:

```
A = Matrix([ [1, 2, -1, 2, 4], [-2, 1, 1, 0, -1], [2, 0, -2, 4, 1] ])
A.rref()[0]
```

Output: [[1, 0, 0, -2, 9/4], [0, 1, 0, 0, 7/4], [0, 0, 1, -4, 7/4]] (Expanded)

The output encodes another system of equations. The first is $1w+0x+0y-2z = 9/4$. This system has two important properties. First, it has the same solutions as the original. Second, the solutions can be read directly from it. 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. The solution is displayed.

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

Eigenvalues and eigenvectors. Try

```
M = Matrix([ [5,-10,-5], [2,14,2], [-4,-8,6] ])
pprint(M.eigenvects())
```

There are two eigenvalues. The second is 10. Its eigenspace has geometric multiplicity two. A basis for the space is shown.

Note the "pprint" (pretty print). Try `GramSchmidt(L)` to see why it is named that.

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

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

Orthogonal output vectors $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\begin{bmatrix} \frac{4}{5} \\ \frac{-2}{5} \end{bmatrix}$ with the same span as $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\begin{bmatrix} 3 \\ 4 \end{bmatrix}$.

`pprint(GramSchmidt(L, True))` returns normalized eigenvectors.

Correlation. Implement vectors as matrices:

```
u = Matrix([1,2,3])
v = Matrix([4,5,6])
correlation(u,v) # Output: 1
```

4.2 Geometric Algebra

Read Section 3, Output, before proceeding here.

Open g3.ipynb. Execute the first cell. It produces no output but gets the geometric algebra $\mathbb{G}^3$ up and running.

Add a new cell:

```
A = y*ex + 3*ex*ey
B = x*ey
gprint(A*B)
```

Output: $3xe_x + xye_x \wedge e_y$.

Substitute. Sometimes you want to substitute specific values for variables. Example: `(A*B).subs({x:1,y:2})` produces $3\mathbf{e}_x + 2\mathbf{e}_x \wedge \mathbf{e}_y$. Note that `subs` leaves variables unchanged, e.g., `A` is unchanged after `gprint(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, i.e., 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.[3] They are given in precedence order (imposed by Python), high to low.[4] Plus and minus are grouped because they have the same precedence.

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

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.

[3] Use "$<$" for the product called the inner product and denoted "$\cdot$" in my books and this document. It is usually called the *left contraction* and denoted $\rfloor$. (The *right contraction* is denoted $\lfloor$.) Use "$|$" for the product called the inner product and denoted "$\cdot$" in, e.g., books by Hestenes and Sobczyk and by Doran and Lasenby.

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

General Multivectors

We have seen that $\mathcal{GA}$lgebra enables assignments of variables to *specific* multivectors, as in `A` above. $\mathcal{GA}$lgebra can also create *general* multivectors. Example:

```
V = g3.mv('V', 'vector')
gprint(V)
```
Output: $V^x\mathbf{e}_x + V^y\mathbf{e}_y + V^z\mathbf{e}_z$

You see that V is a multivector, a general vector in $\mathbb{G}^3$. This is different from a SymPy vector.

The first parameter, `V`, of `g3.mv` is a string, the output name of the variable V. An optional third parameter `f=True` makes the multivector a function of the coordinates, i.e., a field. 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)
'odd'	odd multivector
'mv'	general multivector

In addition, `g3.mv(c)` is available, where c is a specific scalar (such as 1).

The scalar result in the top row is a general scalar multivector. This is different from a SymPy scalar.

Multivector Methods

To use (most of) these, you need `from galgebra.mv import *`.

In the list below, `M` is a multivector; `A, B` are blades; `u` and `v` 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.
`Nga(M, prec=k)`	Returns M to k significant figures.
`M.dual()`	M^*. Returns MI. My books use $M^* = MI^{-1}$. For this, issue `Ga.dual_mode('Iinv-')` after imports.
`M.even()`	Even grades of M
`M.exp()`	e^M. M^2 must be a scalar constant. If $M^2 > 0$, use `exp(M,'+')`
`M.grade(r)`	$\langle M \rangle_r$
`M.grade()`	$\langle M \rangle_0$
`M.inv()`	M^{-1}
`M.norm()`	$\|M\|$
`M.norm2()`	$\|M\|^2$
`M.odd()`	Odd grades of M
`M.rev()`	$M^\dagger$. Reverse of M.
`proj(B,A)`	$\mathsf{P}_{\mathbf{B}}(\mathbf{A})$. Projection onto blade $\mathbf{B}$ of $\mathbf{A}$.
`rot(itheta,A)`	$\mathsf{R}_{i\theta}(\mathbf{A})$. Rotation by angle $i\theta$ of $\mathbf{A}$.
`refl(B,A)`	$\mathsf{M}_{\mathbf{B}}(\mathbf{A})$. Reflection in blade $\mathbf{B}$ of $\mathbf{A}$.
`ga.com(A,B)`	Geometric algebra commutator: $[A, B] = \frac{1}{2}(AB - BA)$
`cross(u,v)`	Cross product. 3D only.

Outermorphisms

You will need `from galgebra.lt import *`.

The following examples create an outermorphism `L` on the geometric algebra g2.

First, a *specific* outermorphism:
`L = g2.lt([[0,2],[1,1]])` or `L = g2.lt([2*ey,ex+ey])` (They are equivialent.)
Then $L(e_x) = 2e_y$ and $L(e_y) = e_x + e_y$).
`gprint(L(ex^ey))` Output: $-2e_x \wedge e_y$.

Now a *general* outermorphism:
`L = g2.lt('L')` Then $L(e_x) = L_{xx}e_x + L_{yx}e_y$, etc.

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

Optional parameters `mode = 's'` or `mode = 'a'` will generate a general symmetric or antisymmetric (skew) matrix, respectively.

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

These can be combined: `L.adj().matrix()`

5 Calculus

5.1 Vector Calculus

This Section 5.1 uses SymPy's vector calculus capabilities, with no geometric calculus involved. The next Section 5.2 will take up $\mathcal{GA}$lgebra's geometric calculus capabilities.

Differentiation, including partial differentiation.
`x,y = symbols('x y', real=True)`
`diff(y*x**2, x)` Output: $2xy$
`diff(diff(y*x**2,x),y)` Output: $2x$
`f = y*x**2`
`diff(f,x)` Output: $2xy$

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', real=True)
X = Matrix([r, theta])
Y = Matrix([r*cos(theta), r*sin(theta)])
Y.jacobian(X)         # Print 2 × 2 Jacobian matrix
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, `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 = symbols('x', real=True)`
`integrate(x**2 + x + 1, x)` Output: $\frac{x^3}{3} + \frac{x^2}{2} + x$.

Iterated integrals. This code evaluates $\int_{x=0}^{1}\int_{y=0}^{1-x}(x+y)\,dy\,dx$:
```
x, y = symbols('x y', real=True)
I1 = integrate(x + y, (y, 0, 1-x))
I2 = integrate(I1, (x, 0, 1))
```

evalf.
`(log(10), log(10).evalf(4))`
Output: (log(10), 2.303)

5.2 Geometric Calculus

Create a multivector field. `F = g3.mv('F', 'vector', f=True)`.

"`f = True`" makes the vector F a function of the coordinates, i.e., a field.

Then $F(x,y,z) = F^x(x,y,x)e_x + F^y(x,y,x)e_y + F^z(x,y,x)e_z$.

Recall `gFormat`:

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

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

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

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

Differential operators. The basic differential operator in $\mathcal{GA}$lgebra is the partial derivative operator.

```
from galgebra.dop import *
```

`g3.Pdop(x)` Output: $\frac{\partial}{\partial x}$.

You can assign: `pdx = g3.Pdop(x)`.

A generalization: `g3.Pdop({x:1,y:2})`. Output: $\frac{\partial^3}{\partial x \partial y^2}$.

Add (+), subtract (-), and multiply/compose (*) these with each other and with multivector functions in any way that makes sense.

Example: `ex*pdx - ey*pdy*(x**2 + y**2)` has output $-2ye_y + e_x\frac{\partial}{\partial x}$.

Let `D` be a differential operator and `A` and `B` be multivector functions. In general, `D*A*B` is not associative: `(D*A)*B` $\neq$ `D*(A*B)`. Moral: Put in parentheses.

Curvilinear coordinates. Curvilinear coordinates are implemented by creating an appropriate geometric algebra. For example, sp3.ipynb creates sp3, a geometric algebra for $\mathbb{R}^3$, using spherical coordinates. Then the gradient operator in sp3 is

$$\text{sp3.grad} = \mathbf{e}_r\partial_r + \mathbf{e}_\phi r^{-1}\partial_\phi + \mathbf{e}_\theta(r\sin\phi)^{-1}\partial_\theta.$$

The notebook sp2.ipynb defines a geometric algebra for the unit sphere in $\mathbb{R}^3$ using spherical coordinates.

Submanifolds. The example sp2g3.ipynb creates a geometric algebra for the unit sphere sp2 in $\mathbb{R}^3$ in spherical coordinates as a submanifold of the geometric algebra g3 in cartesian coordinates. (g3 is defined within sp2g3.ipynb.) Then `sp2.grad` $= \mathbf{e}_\phi\partial_\phi + \mathbf{e}_\theta(\sin\phi)^{-1}\partial_\theta$, the gradient operator of `sp2`. It is the restriction of `sp3grad` to the sphere.

The example sp2sp3.ipynb creates the same geometric algebra sp2, but this time as a submanifold of the geometric algebra sp3 in spherical coordinates from Section 4.2. (sp3 is defined within sp2sp3.ipynb). Of course `sp2.grad` is as before.