$\mathcal{GA}$lgebra Primer

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

November 29, 2022

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.) Gregory Grunberg is active in both improving and finding bugs in $\mathcal{GA}$lgebra. And Greg has made several good suggestions about this primer.

The Python package SymPy (Symbolic Python) is a computer algebra system written in the popular programming language Python. It provides *symbolic* computation capabilities, similar to Mathematica and Maple. For example, it can invert symbolic matrices. (See p. 8.)

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

My intent is that to do the exercises and problems in my books requiring $\mathcal{GA}$lgebra you need not know more programming than is documented here.

Contents

1 Install

• Install the latest version of Python 3 (*not* Python 2).
Download and execute the Python installation file.
Documentation. Tutorial. These are for reference only. I am not recommending that you study them.

Python 3 includes the package manager *pip*. Use it below to install SymPy, JupyterLab, and $\mathcal{GA}$lgebra. Run pip from any folder's command line. To show pip's options, type "pip".

• Install SymPy: `pip install sympy`. Also: `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 JupyterLab: `pip install jupyterlab`.[1]
JupyterLab home, Getting started, Documentation, Wikipedia.

JupyterLab is a *notebook environment*: it runs in a web browser and provides *interactive* Python programming capability. Notebook files have an ipynb extension. I keep mine in one folder, the "root", and its subfolders. JupyterLab allows navigation only in the root and it subfolders – something about security.
You can have more than one root folder.

• Download GAfiles.zip. Unzip it into your root folder. Locate the folder "galgebra", a subfolder of "site-packages" in your Python distribution, where ga.py is. Copy the files gprinter.py, lt.py, mv.py, and GAlgebraInit.py from the root folder to the "galgebra" folder. You might have to give permission to overwrite lt.py and mv.py.
Most other files in your root folder have an ipynb extension. They define a specific geometric algebra. For example g3.ipynb defines the geometric algebra $\mathbb{G}^3$. See Section 4.2.

If you are connected to the internet, your installation is complete. LaTeX output will be processed by MathJax. Otherwise you need to install a LaTeX system.[2,3]

[1] I had a report WARNING of a Jupyter Lab installation problem from an Ubuntu user: Do not use conda to install the recommended software; use pip as recommended. Add the path to Jupyter Lab to the bottom of your .bashrc file:

export PATH=$PATH:/home/USERNAME/.local/bin

Replace USERNAME with your username (the correct path is usually given in a warning by pip at the end of the installation).

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

[3] There are many distributions of Python with extra bells and whistles. Anaconda is large (5 GB), free, multiplatform, and popular. Anaconda includes SymPy and JupyterLab. GAlgebra must be installed separately.

I often find Google more help than the documentation links above. For example, when I wanted to know how to make the small matrix on p. 6, I Googled "latex small matrix".

Create a shortcut (Windows), alias (Mac OS), or symlink (Linux) to JupyterLab

Windows.
• Create a shortcut to jupyter-lab.exe in the Scripts folder of your Python installation: Hold down Shift-Ctrl, drag jupyter-lab.exe to the desktop, and release it.
• Right click on the shortcut, select "Properties", and enter the location of your root in "Start in".
• Rename the shortcut if you like.
• I pinned it to my taskbar.

Click on the shortcut and JupyterLab will open, displaying your root in the left sidebar. If you don't see this, use the "View menu".

Mac OS.[4]
• Click on the Finder icon at the bottom left of the screen.
• Right-click on JupyterLab on the left side of the window.
• Select "Make Alias". An alias will be created in the same older as JupyterLab.
• Click "Enter" and drag the alias to your desktop. (Use the "Enter" key on the numeric keypad or at the bottom of the keyboard, not the "Enter" or "Return" key to the right of the single and double quotes key.) You can also drag the alias to the applications area of the Dock, on the left.

Linux.[5] Hold Shift+Ctrl and drag the file or folder you want to link to to the location where you want the shortcut. This method may not work with all desktop managers.

In a terminal: "ln -s [/folderorfile/link/will/point/to] JupyterLab".

[4] I thank Arthur Herring for help with this.
[5] From https://www.faqforge.com/linux/create-shortcuts-in-linux-symbolic-links

2 JupyterLab

This section will get you get started with JupyterLab. 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 JupyterLab.

JupyterLab's help menu includes help for Python, SymPy, and JupyterLab.

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 see a list of shortcuts: *Settings* → *Advanced Settings Editor* or "Ctrl ,".

Copy and paste. Whole cells: Select a cell or contiguous cells. Then copy and paste – even between different notebooks – using shortcuts or the Edit menu. Parts of cells: Copy and paste 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.

Return to Root Folder from Subfolder. Click on small folder icon above "Name" toward upper left.

Code and Markdown cells. A JupyterLab notebook can contain *code cells* for executable code (labeled to their left by "[n]:"), and *markdown cells* for commentary on the notebook. The toolbar (below the tabs) enables change from one type to another. Documentation Documentation

The help menu offers a 10 minute markdown tutorial.

JupyterLab programming tips.

- You can go back to an earlier cell, modify it if you like, and reexecute it.
- Break your program into cells of reasonable size.
- Document your code in markdown cells

Issue terminal commands in JupyterLab. Type a "!" followed by the command in a JupyterLab cell and execute it. Example: `!pip install numpy`.

3 Output

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

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

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 to define 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, using 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.

`(exr, eyr, ezr) = g3.mvr()` Program names of reciprocal basis vectors. Optional. Equal to `(ex, ey, ez)` for an orthonormal basis.

Execute Cell 3. `gprint` is $\mathcal{GA}$lgebra's output statement. It 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 using 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	$\wedge$
		$*$
∇	\grad	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 the notebook programs from GAfiles.zip are identical. The cell initializes Sympy and $\mathcal{GA}$lgebra, getting them ready for use in the program. Start every program with this cell. The second cell of each notebook defines a geometric algebra.

GAlgebraInit.py from GAfiles.zip can also initialize Sympy and $\mathcal{GA}$lgebra. To use it execute `from galgebra.GAlgebraInit import *` in the first cell of a program. After initialization, GAlgebraInit announces the current versions of Python, SymPy, and $\mathcal{GA}$lgebra.

GAlgebraInit saves typing. More important, if the initialization needs to be changed, as is possible, you need not change all your programs, but only GAlgebraInit.

Fmt. Specifies how a multivector M is split across lines when output:

`Fmt(1)`: All of M is output on one line. Default.

`Fmt(2)`: Each grade of M is output on a separate line.

`Fmt(2)`: Each term of M is output on a separate line.

A `Fmt(n)` remains in effect until another is issued.

gFormat. Use one or both parameters for functions f.

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

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

Warning: multiple gprints in a cell. A cell with many gprints can take seconds to produce output on my reasonably powerful PC.

Notation. We 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 appear in `this font`.

In the examples below I can mostly cut from this pdf and paste into a JupyterLab 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 in JupyterLab. Section 4.2 takes up $\mathcal{GA}$lgebra and geometric algebra in JupyterLab.

Open JupyterLab. 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 from this document. Anything after a "#" is a comment, so need not be copied.

```
from sympy import *  # Make SymPy available to the program.
```

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 SymPy 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 SymPy *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, then look up particular topics when you need them.

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

Span (rref). 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 (LDLsolve).

```
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).
```

Singular value decomposition (svd). The svd used here is from the package NumPy (Numerical Python). (As of this writing there is no *symbolic* SVD in SymPy.) Install the package with pip if you haven't done this already. The following program defines a matrix `A`, performs an svd on it, puts it back together, and prints it.

```
import numpy as np
np.set_printoptions(precision=14)
A = np.array([[7, 3, 5], [2, 4, 3]]) # Define matrix to be SVDed.
U, S, Vt = np.linalg.svd(A, full_matrices=False) # Perform SVD
print(U @ np.diag(S) @ Vt) # @ is matrix multiplication
```

The original `A` is exactly recovered to 14 decimal places. Increase this to 15 and a rounding error appears.

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). *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`:

```
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 (eigenvects). 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 (GramSchmidt). 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 (correlation). Implement vectors as matrices:

```
from galgebra.mv import correlation
u = Matrix([1,2,3,4,5])
v = Matrix([1,3,3,7,12])
correlation(u,v,dec=3) # Output: .938
```

Highly, but not exactly, correlated.

4.2 Geometric Algebra

Read Section 3, Output, before proceeding here.

Open g3.ipynb.
Execute Cell 1. It gets $\mathcal{GA}$lgebra up and running. No output is produced.
Execute Cell 2. It defines the geometric algebra $\mathbb{G}^3$. No output is produced.

Add a new cell and execute:

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

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

3 is a SymPy scalar, x and y are SymPy symbols, and e_x and e_y are the printing names of the noncommuting SymPy symbols ex and ey from cell 2.

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*: multiplication comes before addition. We say that 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.[6] They are given in precedence order (imposed by Python), high to low.[7] 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.

Remember (the very unfortunate) need for parentheses around terms with inner or outer products. Omitting them is a common error.

[6] Use "$<$" for the product called the inner product and denoted "$\cdot$" in the book by Dorst, Fontijne, and Mann; in my books; and in 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.

[7] Here is 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: $V^x\mathbf{e}_x + V^y\mathbf{e}_y + V^z\mathbf{e}_z$ ($V_x\mathbf{e}_x + V_y\mathbf{e}_y + V_z\mathbf{e}_z$ in my books.)
The superscript x, y, z on the V's is a convention of $\mathcal{GA}$lgebra.
They would appear as subscripts in LAGA.

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

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:

2^{nd}	Result
'scalar'	scalar multivector
'vector'	vector multivector
'bivector'	bivector multivector
n	grade n multivector
'pseudo'	pseudoscalar multivector
'even'	even multivector
'odd'	odd multivector
'mv'	general multivector

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

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`. Most have two forms, e.g, `M.dual()` and `dual(M)`.

`ga.E()`	Outer product of the basis vectors of `ga`.
`ga.I()`	Unit pseudoscalar of `ga`, i.e., `ga.E()` normalized.
`ga.Iinv()`	Inverse of unit pseudoscalar of `ga`.
`ga.com(A,B)`	Geometric algebra commutator: $[A,B] = \frac{1}{2}(AB - BA)$
`Nga(M, prec=k)`	Returns M to k significant figures.
`M.dual()`	$M^* = MI$.
In my books $M^* = MI^{-1}$.	For this, issue `Ga.dual_modes('Iinv+')` after imports.
`M.unual()`	$M^{-*} = MI^{-1}$. The inverse of the dual operator.
In my books $M^{-*} = MI$.	`Ga.dual_modes('Iinv+')` takes care of this also.
`M.even()`	Even grades of M
`M.exp()`	e^M. M^2 must be a scalar constant.
	If $M^2 > 0$, issue `exp(M,'+')` first.
`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 $\mathbf{A}$.
`cross(u,v)`	Cross product. 3D only.

Outermorphisms

First, a *specific* outermorphism:
`L = g2.lt([[0,2],[1,1]])` or, equivalently, `L = g2.lt([2*ey,ex+ey])`
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 $L(e_x) = L^x{}_x e_x + L^y{}_x e_y$, etc. ($L(e_x) = L_{xx} e_x + L_{yx} e_y$ in my books.) An optional parameter `f=True` makes the L's functions of x and y.

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

Outermorphisms can be multiplied by scalars ($*$), added ($+$), subtracted ($-$), and composed ($*$). `L.det()` (determinant), `L.adj()` (adjoint), `L.tr()` (trace), `L.inv()` (inverse), `L.is_singular()`, and `L.matrix()`. are also available.

These can be combined, e.g., `L.adj().matrix()`.

Simplify Expressions

Simplifying an expression is a difficult task for a computer algebra system. For a start, there is no definition of "expression A is simpler than expression B". Sometimes it is a matter of a particular use or of taste. And each kind of function, for example trigonometric, has its own identities, adding to the possibilities for simplification.

The result of a simplification might not be what you expect. It can even be less simple to your eyes than the original!

$\mathcal{GA}$lgebra and SymPy each have a simplify function. $\mathcal{GA}$lgebra's simplify simplifies the coefficients of a multivector. It applies SymPy's `simplify` to each coefficient, one at a time. SymPy does the actual work. (The link above, like all on this page, is to a tutorial on the function.)

SymPy's simplify applied to a SymPy scalar illustrates this most simply.

`simplify(sin(x)**2 + cos(x)**2)` Output: 1
`(1/x + 1/y).simplify()` Output: $\frac{x+y}{xy}$

Simplify analyzes its input to figure out what to do. That worked well in the examples. But in more complicated situations simplify can be slow, since it tries many kinds of simplifications before picking the "best" one. If you know what kind of simplification you are after, it is better to apply a specific simplification function.

One such function is `trigsimp`, which simplifies an expression using trig identities. Another is `ratsimp`, which puts an expression over a common denominator and cancels. The next examples show the syntax:

Suppose that `M` is a multivector requiring `trigsimp` and `N` is a multivector requiring `ratsimp`: `simplify(M, modes=trigsimp)` `(N.simplify(modes=ratsimp)`.

$\mathcal{GA}$lgebra's simplify can specify several simplification functions in succession. Suppose that `M` is a multivector requiring both `trigsimp` and `ratsimp` simplifications:

```
simplify(M, modes=[trigsimp,ratsimp])
M.simplify(modes=[trigsimp,ratsimp])
```

More modes for simplification:

`factor` factors a polynomial.
`expand` expands polynomial expressions.
`collect` collects common powers of a term in an expression.
`cancel` cancels common factors in a quotient of polynomials.
`powsimp` $x^a x^b \to x^{a+b}$ and $x^a y^a \to (xy)^a$.
`logcombine` $\log(x) + \log(y) \to \log(xy)$ and $n\log(x) \to \log(x^n)$.
`fu` is another SymPy trigonometric simplification function.
It works better than trigsimp but is more computationally expensive. Complicated!

In fact, $\mathcal{GA}$lgebra's simplify can be used to apply any scalar *sympy* function to the coefficients of a multivector.

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 $\int f(x)\,dx$.
`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 name) 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 `Pdop`.

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

`Pdop(x)` Output: $\frac{\partial}{\partial x}$. `x` must be a declared in a `symbols` statement.

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

A generalization: `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: Use 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.