Kingdon Documentation

Release 1.0.4

Martin Roelfs

Apr 30, 2024

CONTENTS:

Try kingdon in your browser

CHAPTER

ONE

KINGDON

Pythonic Geometric Algebra Package

- Free software: MIT license
- Documentation: https://kingdon.readthedocs.io.

Try kingdon in your browser

1.1 Features

Kingdon is a Geometric Algebra (GA) library which combines a Pythonic API with symbolic simplification and just-in-time compilation to achieve high-performance in a single package. It support both symbolic and numerical GA computations. Moreover, `kingdon` uses `ganja.js` for visualization in notebooks, making it an extremely well rounded GA package.

In bullet points:

- Symbolically optimized.
- Leverage sparseness of input.
- `ganja.js` enabled graphics in jupyter notebooks.
- Agnostic to the input types: work with GA's over `numpy` arrays, `PyTorch` tensors, `sympy` expressions, etc. Any object that overloads addition, subtraction and multiplication makes for valid multivector coefficients in `kingdon`.
- Automatic broadcasting, such that transformations can be applied to e.g. point-clouds.
- Compatible with `numba` and other JIT compilers to speed-up numerical computations.

1.2 Code Example

In order to demonstrate the power of `Kingdon`, let us first consider the common use-case of the commutator product between a bivector and vector.

In order to create an algebra, use `Algebra`. When calling `Algebra` we must provide the signature of the algebra, in this case we shall go for 3DPGA, which is the algebra $\mathbb{R}_{3,0,1}$. There are a number of ways to make elements of the algebra. It can be convenient to work with the basis blades directly. We can add them to the local namespace by calling `locals().update(alg.blades)`:

```
>>> from kingdon import Algebra
>>> alg = Algebra(3, 0, 1)
>>> locals().update(alg.blades)
>>> b = 2 * e12
>>> v = 3 * e1
>>> b * v
-6 2
```

This example shows that only the `e2` coefficient is calculated, despite the fact that there are 6 bivector and 4 vector coefficients in 3DPGA. But by exploiting the sparseness of the input and by performing symbolic optimization, `kingdon` knows that in this case only `e2` can be non-zero.

1.3 Symbolic usage

If only a name is provided for a multivector, `kingdon` will automatically populate all relevant fields with symbols. This allows us to easily perform symbolic computations.

```
>>> from kingdon import Algebra
>>> alg = Algebra(3, 0, 1)
>>> b = alg.bivector(name='b')
>>> b
b01 01 + b02 02 + b03 03 + b12 12 + b13 13 + b23 23
>>> v = alg.vector(name='v')
>>> v
v0 0 + v1 1 + v2 2 + v3 3
>>> b.cp(v)
(b01*v1 + b02*v2 + b03*v3) 0 + (b12*v2 + b13*v3) 1 + (-b12*v1 + b23*v3) 2 + (-b13*v1 -␣
↪b23*v2) 3
```

It is also possible to define some coefficients to be symbolic by inputting a string, while others can be numeric:

```
>>> from kingdon import Algebra, symbols
>>> alg = Algebra(3, 0, 1)
>>> b = alg.bivector(e12='b12', e03=3)
>>> b
3 03 + b12 12
>>> v = alg.vector(e1=1, e3=1)
>>> v
1 1 + 1 3
>>> w = b.cp(v)
>>> w
3 0 + (-b12) 2
```

A `kingdon` MultiVector with symbols is callable. So in order to evaluate `w` from the previous example, for a specific value of `b12`, simply call `w`:

```
>>> w(b12=10)
3 0 + -10 2
```

1.3.1 Overview of Operators

Table 1: Operators

Operation	Expression	Infix	Inline
Geometric product	ab	`a*b`	`a.gp(b)`
Inner	$a \cdot b$	`a|b`	`a.ip(b)`
Scalar product	$\langle a \cdot b \rangle_0$	`(a|b).grade(0)`	`a.sp(b)`
Left-contraction	$a \rfloor b$		`a.lc(b)`
Right-contraction	$a \lfloor b$		`a.rc(b)`
Outer (Exterior)	$a \wedge b$	`a ^ b`	`a.op(b)`
Regressive	$a \vee b$	`a & b`	`a.rp(b)`
Conjugate `a` by `b`	$ba\widetilde{b}$	`b >> a`	`b.sw(a)`
Project `a` onto `b`	$(a \cdot b)\widetilde{b}$	`a @ b`	`a.proj(b)`
Commutator of `a` and `b`	$a \times b = \frac{1}{2}[a, b]$		`a.cp(b)`
Anti-commutator of `a` and `b`	$\frac{1}{2}\{a, b\}$		`a.acp(b)`
Sum of `a` and `b`	$a + b$	`a + b`	`a.add(b)`
Difference of `a` and `b`	$a - b$	`a - b`	`a.sub(b)`
Reverse of `a`	$\widetilde{a}$	`~a`	`a.reverse()`
Squared norm of `a`	$a\widetilde{a}$		`a.normsq()`
Norm of `a`	$\sqrt{a\widetilde{a}}$		`a.norm()`
Normalize `a`	$a/\sqrt{a\widetilde{a}}$		`a.normalized()`
Square root of `a`	$\sqrt{a}$		`a.sqrt()`
Dual of `a`	$a*$		`a.dual()`
Undual of `a`			`a.undual()`
Grade `k` part of `a`	$\langle a \rangle_k$		`a.grade(k)`

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install Kingdon, run this command in your terminal:

```
$ pip install kingdon
```

This is the preferred method to install Kingdon, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Kingdon can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/tbuli/kingdon
```

Or download the tarball:

```
$ curl -OJL https://github.com/tbuli/kingdon/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

BASIC USAGE

The most important object in all of `kingdon` is `Algebra`:

```
from kingdon import Algebra
```

An `Algebra` has to be initiated with a number of positive, negative, and null-dimensions, which are traditionally denoted by `p`, `q` and `r`. For example, in order to create a 2D Geometric Algebra we can initiate

```
>>> alg = Algebra(p=2, q=0, r=0)
>>> alg = Algebra(2)  # Equivalent: default value for p, q, r is 0.
```

The basis blades of the algebra are available in the dictionary `alg.blades`. This can be added to `locals()` in order to allow for easy access to all the basis blades, and allows the initiation of multivectors using the basis-blades directly:

```
>>> locals().update(alg.blades)
>>> x = 2 * e + 1 * e1 - 5 * e2 + 6 * e12
```

where `e` is the identity element, i.e. $e = 1$. This way of creating multivectors is particularly useful when writing quick scripts in an interactive environment. Let's look at some more general ways of making multivectors, starting with symbolic multivectors before we go on to numerical multivectors.

3.1 Symbolic Multivectors

In order to create symbolical multivectors in an algebra, we can call *`multivector`* and explicitly pass a `name` argument. For example, let us create two symbolic vectors `u` and `v` in this algebra:

```
>>> u = alg.multivector(name='u', grades=(1,))
>>> v = alg.multivector(name='v', grades=(1,))
>>> u
(u1) * e1 + (u2) * e2
>>> v
(v1) * e1 + (v2) * e2
```

The return type of *`multivector()`* is an instance of *`MultiVector`*.

Note: `kingdon` offers convenience methods for common types of multivectors, such as the vectors above. For example, the vectors above can also be created using `u = alg.vector(name='u')`. Moreover, a scalar is created by *`scalar()`*, a bivector by *`bivector()`*, a pseudoscalar by *`pseudoscalar()`*, and so on. However, all of these merely add the corresponding `grades` argument to your input and then call *`multivector`*, so *`multivector`* is what we need to understand.

MultiVector's support common math operators:

```
>>> u + v
(u1 + v1) * e1 + (u2 + v2) * e2
>>> u * v
(u1*v1 + u2*v2) + (u1*v2 - u2*v1) * e12
```

We also have the inner and exterior "products":

```
>>> u | v
(u1*v1 + u2*v2)
>>> u ^ v
(u1*v2 - u2*v1) * e12
```

We see that *in the case of vectors* the product is equal to the sum of the inner and exterior.

Since vectors in 2DVGA represent reflections in lines through the origin, we can reflect the line `v` in the line `u` by using conjugation:

```
>>> u >> v
(u1**2*v1 + 2*u1*u2*v2 - u2**2*v1) * e1 + (-u1**2*v2 + 2*u1*u2*v1 + u2**2*v2) * e2
```

we see that the result is again a vector, as it should be.

These examples should show that the symbolic multivectors of `kingdon` make it easy to do symbolic computations. Moreover, we can also use `sympy` expressions as values for the multivector:

```
>>> from sympy import Symbol, sin, cos
>>> t = Symbol('t')
>>> x = cos(t) * e + sin(t) * e12
>>> x.normsq()
1
```

3.1.1 More control over basisvectors

If we do not just want to create a symbolic multivector of a certain grade, but with specific blades, we can do so by providing the `keys` argument.

```
>>> x = alg.multivector(name='x', keys=('e1', 'e12'))
>>> (x1) * e1 + (x12) * e12
```

This can be done either by providing a tuple of strings which indicate which basis-vectors should be present, or by passing them as integers, i.e. `keys=(0b01, 0b11)` is equivalent to the example above. Internally, `kingdon` uses the binary representation.

3.2 Numerical Multivectors

While `kingdon` makes no assumptions about the data structures that are passed into a multivector in order to support ducktyping and customization as much as possible, it was nonetheless designed to work really well with `numpy` arrays.

For example, to repeat some of the examples above with numerical values, we could do

```
>>> import numpy as np
>>> uvals, vvals = np.random.random((2, 2))
>>> u = alg.vector(uvals)
>>> v = alg.vector(vvals)
>>> u * v
(0.1541) + (0.0886) * e12
```

A big performance bottleneck that we suffer from in Python, is that arrays over objects are very slow. So while we could make a numpy array filled with `~kingdon.multivector.MultiVector`'s, this would tank our performance. `kingdon` gets around this problem by instead accepting numpy arrays as input. So to make a collection of 3 lines, we do

```
>>> import numpy as np
>>> uvals = np.random.random((2, 3))
>>> u = alg.vector(uvals)
>>> u
([0.82499172 0.71181276 0.98052928]) * e1 + ([0.53395072 0.07312351 0.42464341]) * e2
```

what is important here is that the first dimension of the array has to have the expected length: 2 for a vector. All other dimensions are not used by `kingdon`. Now we can reflect this multivector in the `e1` line:

```
>>> v = alg.vector((1, 0))
>>> v >> u
([0.82499172 0.71181276 0.98052928]) * e1 + ([-0.53395072 -0.07312351 -0.42464341]) * e2
```

Despite the different shapes, broadcasting is done correctly in the background thanks to the magic of numpy, and with only minor performance penalties.

3.3 Operators

Instances of *`MultiVector`* overload all common Geometric Algebra operators. Below is an overview:

Table 1: Operators

Operation	Expression	Infix	Inline
Geometric product	ab	`a*b`	`a.gp(b)`
Inner	$a \cdot b$	`a|b`	`a.ip(b)`
Scalar product	$\langle a \cdot b \rangle_0$	`(a|b).grade(0)`	`a.sp(b)`
Left-contraction	$a \rfloor b$		`a.lc(b)`
Right-contraction	$a \lfloor b$		`a.rc(b)`
Outer (Exterior)	$a \wedge b$	`a ^ b`	`a.op(b)`
Regressive	$a \vee b$	`a & b`	`a.rp(b)`
Conjugate a by b	$ba\widetilde{b}$	`b >> a`	`b.sw(a)`
Project a onto b	$(a \cdot b)\widetilde{b}$	`a @ b`	`a.proj(b)`
Commutator of a and b	$a \times b = \frac{1}{2}[a, b]$		`a.cp(b)`
Anti-commutator of a and b	$\frac{1}{2}\{a, b\}$		`a.acp(b)`
Sum of a and b	$a + b$	`a + b`	`a.add(b)`
Difference of a and b	$a - b$	`a - b`	`a.sub(b)`
Reverse of a	$\widetilde{a}$	`~a`	`a.reverse()`
Squared norm of a	$a\widetilde{a}$		`a.normsq()`
Norm of a	$\sqrt{a\widetilde{a}}$		`a.norm()`
Normalize a	$a/\sqrt{a\widetilde{a}}$		`a.normalized()`
Square root of a	$\sqrt{a}$		`a.sqrt()`
Dual of a	$a*$		`a.dual()`
Undual of a			`a.undual()`
Grade k part of a	$\langle a \rangle_k$		`a.grade(k)`

Note that formally conjugation is defined by bab^{-1} and projection by $(a \cdot b)b^{-1}$, but that both are implemented using reversion instead of an inverse. This is because reversion is much faster to calculate, and because in practice b will often by either a rotor satisfying $b\widetilde{b} = 1$ or a blade satisfying $b^2 = b \cdot b$, and thus the inverse is identical to the reverse (up to sign).

If you want to replace these operators by their proper definitions, you can use the register decorator to overwrite the default operator (use at your own risk):

```
>>> @alg.register(name='sw')
>>> def sw(x, y):
>>>     return x * y / y
>>> @alg.register(name='proj')
>>> def proj(x, y):
>>>     return (x | y) / y
```

3.4 Graphing using `ganja.js`

`kingdon` supports the `ganja.js` graphing syntax. For those already familiar with `ganja.js`, the API will feel very similar:

```
>>> alg.graph(0xff0000, u, "u", lineWidth=3)
```

The rules are simple: all positional arguments will be passed on to `ganja.js` as elements to graph, whereas keyword arguments are passed to `ganja.js` as options. Hence, the example above will graph the line `u` with `lineWidth = 3`, and will attach the label "u" to it, and all of this will be red. Identical to `ganja.js`, valid inputs to `alg.graph` are (lists

of) instances of `MultiVector`, strings, and hexadecimal numbers to indicate colors, or a function without arguments that returns these things. The strings can be simple labels, or valid SVG syntax.

Note: kingdon supports `ganja.js`'s animation and interactivity in jupyter notebooks, try kingdon in your browser to give it a go!

3.5 Performance Tips

Because `kingdon` attempts to symbolically optimize expressions using `sympy` the first time they are called, the first call to any operation is comparatively slow, whereas subsequent calls have very good performance.

There are however several things to be aware of to ensure good performance.

3.5.1 Graded

The first time `kingdon` is asked to perform an operation it hasn't seen before, it performs code generation for that particular request. Because codegen is the most expensive step, it is beneficial to reduce the number of times it is needed. An easy way to achieve this is to initiate the `Algebra` with *graded=True*. This enforces that `kingdon` does not specialize codegen down to the individual basis blades, but rather only per grade. This means there are far less combinations that have to be considered and generated.

3.5.2 Numba JIT

We can enable numba just-in-time compilation by initiating an `Algebra` with *wrapper=numba.njit*. This comes with a significant cost the first time any operator is called, but subsequent calls to the same operator are significantly faster. It is worth mentioning that when dealing with *Numerical Multivectors* over numpy arrays, the benefit of using *numba* actually reduces rapidly as the numpy arrays become larger, since then most of the time is spend in numpy routines anyway.

3.5.3 Register Expressions

To make it easy to optimize larger expressions, `kingdon` offers the `register()` decorator.

```
>>> alg = Algebra(3, 0, 1)
>>>
>>> @alg.register
>>> def myfunc(u, v):
>>>     return u * (u + v)
>>>
>>> x = alg.vector(np.random.random(4))
>>> y = alg.vector(np.random.random(4))
>>> myfunc(x, y)
```

Calling the decorated `myfunc` has the benefit that all the numerical computation is done in one single call, instead of doing each binary operation individually. This has the benefit that all the (expensive) python boilerplate code is called only once.

CHAPTER

FOUR

INNER WORKINGS

This chapter explains how `kingdon` works internally.

CHAPTER

FIVE

EXAMPLES

Try kingdon in your browser

5.1 2DPGA

5.1.1 Symbolic Example

[1]:
```
from kingdon import Algebra
```

Let us create two symbolic vectors u and v.

[2]:
```
alg = Algebra(3, 0, 1)
u = alg.vector(name='u')
v = alg.vector(name='v')
```

Their product is a bireflection, and has a scalar and bivector part:

[3]:
```
R = u * v
print('R =', R)
print('grades:', R.grades)
```

```
R = (u1*v1 + u2*v2 + u3*v3) + (u0*v1 - u1*v0)  + (u0*v2 - u2*v0)  + (u0*v3 - u3*v0)  +␣
↪(u1*v2 - u2*v1)  + (u1*v3 - u3*v1)  + (u2*v3 - u3*v2)
grades: (0, 2)
```

While the product of two different vectors has a non-zero bivector part, the product of a vector with itself only has a scalar part:

[4]:
```
usq = u * u
print('usq =', usq)
print('grades:', usq.grades)
```

```
usq = (u1**2 + u2**2 + u3**2)
grades: (0,)
```

Kingdon has realized this, and has removed the bivector part in the output entirely, and not just set it equal to zero.

To evaluate this square for given numerical values of the coefficients, we can call the symbolic expression:

[5]:
```
import numpy as np
```

(continues on next page)

(continued from previous page)

```
res = usq(u1=np.cos(np.pi/3), u2=np.sin(np.pi/3), u3=0)
res
```

[5]:
```
1.0
```

[ ]:
```

```

5.1.2 Poincloud rotor estimation

Consider the following challenge. We are presented with an input pointcloud p_i, and an output pointcloud $q_i = R[p_i] + \eta_i$, where R is an unknown tranformation, and η_i is Gaussian noise. The challenge is to reconstruct the transformation R.

In order to do this, we construct a symbolic tranformation R, whose entries are `symfit.Parameter` objects. We can then use `symfit` to find the rotor R.

[1]:
```
from kingdon import Algebra
from symfit import Fit, Model, CallableModel, Variable, Parameter, Eq, Mul
from symfit.core.minimizers import *
import numpy as np
```

We set up the number of points `n_points` in the pointcloud, the number of (Euclidean) dimensions of the modeling space `d`, and the standard deviation `sig` of the Gaussian distribution.

[2]:
```
n_points = 10
d = 2
sig = 0.02
```

[3]:
```
point_vals = np.zeros((d+1, n_points))
noise_vals = np.zeros((d+1, n_points))
point_vals[0, :] = np.ones(n_points)
point_vals[1:, :] = np.random.random((d, n_points))
noise_vals[1:, :] = np.random.normal(0.0, sig, (d, n_points))
```

[4]:
```
alg = Algebra(d, 0, 1)
locals().update(alg.blades)
```

Create the points and noise as pseudovector of grade `d`.

[5]:
```
noise = alg.vector(noise_vals).dual()
p = alg.vector(point_vals).dual()
p
```

[5]:
```
[0.6193865  0.05568265 0.10402929 0.62240814 0.63319893 0.57874796
 0.47162778 0.78012503 0.70366406 0.10142499]  + [-0.80124716 -0.13055387 -0.27026671 -0.
↪44351767 -0.98847871 -0.94480035
 -0.07393811 -0.63619458 -0.97730121 -0.76623394]  + [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

As the input rotor R, we use a translation by 0.5 in the $_{20}$ direction, followed by a rotation around $_{12}$.

[6]:
```
t = np.pi/3
T = alg.multivector(e=1, e02=-0.5)
```

(continues on next page)

(continued from previous page)

```
S = alg.multivector(e=np.cos(t), e12=np.sin(t))
R = S*T
print(f'{T=!s}')
print(f'{S=!s}')
print(f'{R=!s}')
```

```
T=1 + -0.5
S=0.5 + 0.866
R=0.5 + -0.433  + -0.25  + 0.866
```

We can now create the transformed pointcloud q, and visualize both both p and q.

[7]:
```
q = R.sw(p).grade(d) + noise
```

[8]:
```
alg.graph(
    0xff0000, p, 'p',
    0x0000ff, q, 'q',
    0x000000, R.grade(2), 'axis',
    scale=1.0,
)
```

[8]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e01', 'e02', 'e12', 'e012'], ['e0', '0',
↪'e01', 'e02', '0', '0', ...
```

We will now setup a symfit model to describe this transformation, where the rotor R consists of `Parameter`'s, and the pointclouds p and q are symfit `Variable`'s.

[9]:
```
R_par = alg.evenmv(name='R', symbolcls=Parameter)
p_var = alg.multivector(name='p', symbolcls=Variable, grades=(d,))
q_var = alg.multivector(name='q', symbolcls=Variable, grades=(d,))
print(R_par)
print(p_var)
print(q_var)
```

```
R + R01  + R02  + R12
p01  + p02  + p12
q01  + q02  + q12
```

[10]:
```
p_var_trans = (R_par >> p_var).filter()
# model = Model({q_var[k]: expr for k, expr in p_var_trans.items()})
model = Model({var: expr for var, expr in zip(q_var.values(), p_var_trans.values())})
model
```

[10]:

$$q_{01}(p_{01}, p_{02}, p_{12}; R, R_{01}, R_{02}, R_{12}) = -RR_{02}p_{12} + R\left(Rp_{01} - R_{02}p_{12} + R_{12}p_{02}\right) + R_{01}R_{12}p_{12} + R_{12}\left(Rp_{02} + R_{01}p_{12} - R_{12}p_{01}\right) \tag{5.1}$$

$$q_{02}(p_{01}, p_{02}, p_{12}; R, R_{01}, R_{02}, R_{12}) = RR_{01}p_{12} + R\left(Rp_{02} + R_{01}p_{12} - R_{12}p_{01}\right) + R_{02}R_{12}p_{12} - R_{12}\left(Rp_{01} - R_{02}p_{12} + R_{12}p_{02}\right) \tag{5.2}$$

$$q_{12}(p_{12}; R, R_{12}) = R^2p_{12} + R_{12}^2p_{12} \tag{5.3}$$

Prepare the data for `symfit`.

[11]:
```
datadict = {var.name: data for var, data in zip(p_var.values(), p.values())}
datadict.update({var.name: data for var, data in zip(q_var.values(), q.values())})
```

Initiate a `symfit.Fit` object with the model and data. We additionally supply the demand $R\widetilde{R} = 1$, since rotors should be normalized (i.e., othonormal transformations).

[12]:
```
constraints = [
    Eq(R_par.normsq().e, 1)
]
fit = Fit(model, **datadict, constraints=constraints)
```

[13]:
```
results = fit.execute()
print(results)
```

```
Parameter Value        Standard Deviation
R         5.026128e-01 9.797416e-03
R01       -4.261785e-01 6.756469e-03
R02       -2.477012e-01 1.336769e-02
R12       8.645116e-01 6.683008e-03
Status message         Optimization terminated successfully
Number of iterations   18
Objective              <symfit.core.objectives.LeastSquares object at 0x0000028B62F303D0>
Minimizer              <symfit.core.minimizers.SLSQP object at 0x0000028B63160E20>

Goodness of fit qualifiers:
chi_squared            0.005224097869669531
objective_value        0.0026120489348347656
r_squared              0.9970428183477802

Constraints:
--------------------
Question: R**2 + R12**2 - 1 == 0?
Answer:   2.525091247207456e-12
```

`symfit` has used SLSQP because of the constraint, and we see that we have high accuracy on this constraint. Let us print the reconstructed rotor and it's norm. Furthermore, we can now apply $\widetilde{R}$ to q to transform it back to the location of p so we can visually inspect the quality of the reconstruction.

[14]:
```
R_re = R_par(**results.params)
print(R_re)
print(R_re.normsq())
```

```
0.503 + -0.426  + -0.248  + 0.865
1.0
```

[15]:
```
p_reconstructed = (~R_re) >> q
```

[16]:
```
from timeit import default_timer

def animate_q():
```

(continues on next page)

(continued from previous page)

```
    """ Make cloud q rotate towards p. """
    s0 = R_re.grade(2).norm().e
    t0 = np.arctan2(s0, R_re.e)
    logR = R_re.grade(2) / s0
    t = t0 * (np.sin(default_timer() / 2) + 1 ) / 2 # [0, t0]

    R = np.cos(t) + logR * np.sin(t)
    return ~R >> q

alg.graph(
    0xff0000, p,
    0x0000ff, q, 'q',
    0x880088, p_reconstructed, 'p reconstructed',
    0x000000, R_re.grade(2), 'reconst. axis',
    animate_q, 'q',
    animate=True,
    scale=1.0,
)
```

[16]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e01', 'e02', 'e12', 'e012'], ['e0', '0',
↪'e01', 'e02', '0', '0', ...
```

We see that we have excellent agreement between the original and reconstructed pointclouds.

[ ]:
```

```

[ ]:
```

```

5.1.3 2DPGA: Points and Lines

This example shows some basic operations on points and lines using 2DPGA.

Based on https://enkimute.github.io/ganja.js/examples/coffeeshop.html#pga2d_points_and_lines.

First we create 2DPGA, and add its basis blades to the local namespace.

[1]:
```
from kingdon import Algebra

alg = Algebra(2, 0, 1)
locals().update(alg.blades)
```

Next, make formulas to construct points and lines from coefficients. In 2D PGA, grade-1 elements or vectors (e0,e1,e2) represent reflections AND lines (the invariant of a reflection).

[2]:
```
line = lambda a, b, c: a*e1 + b*e2 + c*e0
```

Grade-2 elements or bivectors (e01,e02,e12) represent rotations/translations AND points/infinite points (the invariant of a rotation/translation). We define them using the dualisation operator (.dual()) to be independent of choice of basis (e.g. e12 vs e21).

[3]:
```
point = lambda x, y: (e0 + x*e1 + y*e2).dual()
```

Construct the points A, B, and C.

[4]:
```
A = point(-1, -1)
B = point(-1, 1)
C = point(1, 1)
```

Define the line

$$y = 0.5 - x \implies x + y - 0.5 = 0$$

[5]:
```
L = line(1, 1, -0.5)
```

A line can also be defined by JOINING ($\vee$) two points. Mathematically this is done with the VEE operator $\vee$, which in kingdon code is implemented with &. Let us create the line

$$M = C \vee A$$

Moreover, we define the line M as a function so it will update when C or A are dragged:

[6]:
```
M = lambda: C & A
```

Similarly, a point can be defined by MEETING (^) two lines. Again, we define the point D as a function so it will update when L or M change.

[7]:
```
D = lambda: L ^ M
```

[8]:
```
g = alg.graph(
    "Drag A,B,C",   # First label is used as title.
    0xD0FFE1,       # Numbers are colors - use hex!
    [A,B,C],        # render polygon ABC.
    0x882288,       # Set the color to purple.
    [B,C],          # Render line segment from B to C.
    0x00AA88,       # Medium green.
    L, "L", M, "M", # Render and label lines.
    0x224488,       # Set color blue.
    D, "D",         # Intersection point of L and M
    0x008844,       # Set darker green
    A, "A",         # Render point A and label it.
    B, "B",         # Render point B and label it.
    C, "C",         # Render point C and label it.
    lineWidth=3, grid=True, labels=True
)
g
```

[8]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e01', 'e02', 'e12', 'e012'], ['e0', '0',
↪'e01', 'e02', '0', '0', ...
```

[ ]:
```

```

5.1.4 2DPGA: Project and reject

This example shows how to perform projection and rejection, using 2DPGA.

Based on https://enkimute.github.io/ganja.js/examples/coffeeshop.html#pga2d_project_and_reject.

First we create 2DPGA, and add its basis blades to the local namespace.

[1]:
```
from kingdon import Algebra

alg = Algebra(2, 0, 1)
locals().update(alg.blades)
```

Next, make formulas to construct points and lines from coefficients, and to project and reject any elements of geometry.

[2]:
```
# Construct points and lines from coefficients
point = lambda x, y: (e0 + x*e1 + y*e2).dual()
line = lambda a, b, c: a*e1 + b*e2 + c*e0

# The formulas for projection and rejection are the same for all elements of geometry in␣
↪PGA
project = lambda a, b: (a | b) / b
reject = lambda a, b: (a | b)
```

Construct the points A, B, and C, and make the line AC by joining A and C.

[3]:
```
A = point(1, 1)
B = point(-1, 1)
C = point(-1, -1)
AC = A & C
```

[4]:
```
alg.graph(
    0xD0FFE1, [A,B,C],
    0x882288, project(B, AC), "project B onto AC",
    0x882288, project(AC, B), "project AC onto B",
    0x008844, reject(AC, B),  "reject AC from B",
    0x00AA88, AC, "AC",
    0x224488, A, "A", B, "B", C, "C",
    lineWidth=3, grid=1, labels=1
)
```

[4]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e01', 'e02', 'e12', 'e012'], ['e0', '0',
↪'e01', 'e02', '0', '0', ...
```

[ ]:
```

```

5.1.5 2DPGA: Distances and Angles

This example shows how to measure distances and angles, using 2DPGA.

Based on https://enkimute.github.io/ganja.js/examples/coffeeshop.html#pga2d_distances_and_angles.

First we create 2DPGA, and add its basis blades to the local namespace.

[1]:
```
from kingdon import Algebra
import math
import time

alg = Algebra(2, 0, 1)
locals().update(alg.blades)
```

Next, make formulas to construct points and lines from coefficients, and to pcalculate distances and angles.

[2]:
```
# Construct points and lines from coefficients
point = lambda x, y: (e0 + x*e1 + y*e2).dual()
line = lambda a, b, c: a*e1 + b*e2 + c*e0

# Distances between elements in 2D PGA (point to point, point to line),
# are always calculated as the length of their join. (assuming normalization)
# The .e selects the scalar coefficient.
# Use alg.rp instead of x & y such that x and y are allowed to be functions.
def dist(x, y):
    return alg.rp(x, y).normsq().e

# Angles between lines in 2D PGA are calculated using the inner product.
# Again normalized elements are assumed.
def angle(x, y):
    return math.acos(alg.ip(x, y).e)
```

Construct the points A, B, and C. The point A is a function so it can be animated.

[3]:
```
A = lambda: point(-1, -0.5*math.sin(0.5 * time.perf_counter()))
B = point(-1, 1)
C = point(1, 1)
```

Create the three lines connecting these points and normalize them. Be sure to call A first since it is a function.

[4]:
```
AB = lambda: (B & A).normalized()
BC = lambda: (C & B).normalized()
CA = lambda: (C & A).normalized()
```

We then create functions that calculate the distances between the points and return them as labels (strings):

[5]:
```
AtoB  = lambda: f"{dist(A, B):.3f}"
BtoC  = lambda: f"{dist(B, C):.3f}"
CtoA  = lambda: f"{dist(C, A):.3f}"
BtoCA = lambda: f"{dist(B, CA):.3f}"
```

Angles between lines as strings (labels for the points)

[6]:
```
CAtoAB = lambda: f"{angle(CA, AB) * 180 / math.pi:.3f}"
ABtoBC = lambda: f"{angle(BC, AB) * 180 / math.pi:.3f}"
CAtoBC = lambda: f"{angle(CA, BC) * 180 / math.pi:.3f}"
```

[7]:
```
alg.graph(
    "Distances and Angles","Drag the points",       # render title
    0xD0FFE1, [A,B,C],                              # render the triangle
    0x00AA88, AB, AtoB, BC, BtoC, CA, CtoA,         # render edges and lengths.
    0x882288, [B, lambda: B @ CA], BtoCA,         # render projection and dist
    0x224488, A, CAtoAB, B, ABtoBC, C, CAtoBC,      # render verts and angles
    lineWidth=3, grid=True, labels=True, animate=True,
)
```

[7]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e01', 'e02', 'e12', 'e012'], ['e0', '0',
↪'e01', 'e02', '0', '0', ...
```

[ ]:
```

```

5.1.6 Fabrik Inverse Kinematics solver

(Taken from the ganja.js coffeeshop)

In the inverse kinematics problem we need to calculate joint angles of a kinematic chain so its base remains fixed and its tip reaches a given target. This is a highly non-linear problem with many solutions. We implement a solver that tries to minimize the differences on all remaining degrees of freedom.

This algorithm readily translates to 3D, is efficient and very well received by artists in a cg animation context.

First we create an algebra. *Feel free to try 3D instead of 2D!*

[1]:
```
from kingdon import Algebra
import numpy as np
from timeit import default_timer
from functools import wraps

def tonp(func):
    @wraps
    def wrapped_func(*args, **kwargs):
        return np.array(func(*args, **kwargs))
    return func

alg = Algebra(2, 0, 1, wrapper=tonp)
```

We then need to set the number of points in the chain, and we initialize points equally along the chain.

[2]:
```
l = 6
d = 3 / l
points = [alg.vector(e0=1, e1=i * d - 1.5, e2=0, e3=0).dual()
                for i in range(l + 1)]
```

Now we define the actual IK solver. Last point in the chain `c` is the target. We set the tip as the target, then cycle to the base and back restoring original lengths.

[3]:
```
def translator(line, dist):
    """ Translate along the line `line` by a distance `dist`. """
    e, e0 = alg.blades.e, alg.blades.e0
    return 1 - 0.5 * dist * (e0 * line.normalized()*e0.dual())

def inverse_kinematics(c):
    # Run four relaxation steps
    for j in range(4):
        # Set the tip to the target. (this will change the length of the last segment.)
        c[-2] = c[-1]
        # Run backwards to the base and restore the lengths along the chain.
        for i in range(l-2, 0, -1):
            c[i] = translator(c[i] & c[i + 1], d) >> c[i + 1]
        # Loop the other way from base to tip again restoring all lengths.
        for i in range(1, l):
            c[i] = translator(c[i - 1] & c[i], -d) >> c[i - 1]
```

[4]:
```
def graph_func():
    inverse_kinematics(points)
    return [
        0x224488, f"Inverse Kinematics",
        0x008844, *zip(points[1:-1], points[:-2]), # Render line segments [[A,B],[B,C],..
↪]
        0x880088, points[0], "Base",               # Render base
        0x00DD88, *points[1:l],                    # Render joint points. [A,B,C,..]
        0x880088, points[l], "Target",             # Render target in purple
    ]

g = alg.graph(
    graph_func, grid=True, lineWidth=6, labels=True,
)
g
```

[4]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e01', 'e02', 'e12', 'e012'], ['e0', '0',
↪'e01', 'e02', '0', '0', ...
```

[ ]:
```

```

5.1.7 Fivebar Mechanism

[1]:
```
import numpy as np
from kingdon import Algebra
import ipywidgets
```

[2]:
```
alg = Algebra(2, 0, 1)
```

[3]:
```
def point(x, y): return alg.vector(e0=1, e1=x, e2=y).dual()
def plane(a, b, c): return alg.vector(e1=a, e2=b, e0=c)

def exp_np(x):
```

(continues on next page)

(continued from previous page)

```
    xsq = (x|x).grade(0)
    if not xsq:
        return 1 + x
    lsqrt = (-xsq.e)**0.5
    return x * np.sinc(lsqrt / np.pi) + np.cos(lsqrt)

def translator(line, dist):
    e0 = line.algebra.blades['e0']
    return 1 + 0.5 * dist * (e0 * line.normalized() * e0.dual())

def elbow(point1, point2, r1, r2, alternative=False):
    """
    Given two points and two radii, give the intersection point of the
    circles centered on those points with those radii.
    """
    u = point1 & point2
    v = u | point1
    usq_inv = 1 / (u|u).e
    s = 0.5 * (usq_inv * (r1**2 - r2**2) + 1)
    t = ((usq_inv * r1**2) - s**2)**0.5
    point2perp = (1 - point1) / 2**0.5 >> point2
    if alternative:
        return point1 + (point2-point1) * s - (point2perp-point1) *  t
    return point1 + (point2-point1) * s + (point2perp-point1) * t

def fivebar(t, A, B, ac, bd, cd, de, theta_D, pol_A, full_output=False, exp=exp_np):
    # Create C by translating A up, and then rotating it by the desired angle.
    tr_ac = translator(A & B, ac)
    # tr_ac = exp(0.5 * ac * alg.blades['e01'])
    if isinstance(pol_A, (tuple, list)):
        theta_At = sum(t**i * coeff for i, coeff in enumerate(reversed(pol_A)))
    else:
        theta_At = pol_A
    M_A = exp(- 0.5 * A * theta_At)
    C = (M_A * tr_ac) >> A
    # Create D as the elbow of C and B.
    D = elbow(B, C, bd, cd)
    # Create E by translating D up, and then rotating it by the desired angle.
    tr_de = translator(C & D, de)
    # tr_de = exp(0.5 * de * alg.blades['e01'])
    M_D = exp(- 0.5 * theta_D * D)
    E = (M_D * tr_de) >> D
    if full_output:
        return C, D, E
    return E
```

[4]:
```
from timeit import default_timer
from functools import partial

DistSlider = partial(ipywidgets.FloatSlider, min=-10, max=10, step=0.01, readout=True,␣
↪readout_format='.2f')
```

(continues on next page)

(continued from previous page)

```
ac_widget = DistSlider(value=4.329777002067039)
bd_widget = DistSlider(value=-6.418389614876762)

usecase = [point(1.5, 3.0), point(5.0, 5.0), point(8.5, 2.0)]
best = [
    7.522375791897677,
    -1.2297104295438015,
    1.4883454465783046,
    9.65978858247454,
    4.329777002067039,
    -6.418389614876762,
    12.91788027878568,
    10.223852704675853,
    -3.393426372984221,
    0.9098149166712446,
    -3.763919738437536,
    -5.266926540202572
]

Ax, Ay, Bx, By, ac, bd, cd, de, theta_D, tA1, tA2, tA3 = best
# Anchor points
A = point(Ax, Ay);
B = point(Bx, By);
endpoints = []

def graph_func():
    t = default_timer() / 1000
    ac = ac_widget.value
    bd = bd_widget.value
    C, D, E = fivebar(t, A, B, ac, bd, cd, de, theta_D, [1, 0, 0], full_output=True)
    endpoints.append(E)
    if len(endpoints) > 50:
        endpoints.pop(0)

    return [
        # usecase
        *usecase,
        # Anchor points
        0xff0000, A, 'A', B, 'B',
        0x000000, [A, C], [B, D], [C, D], [D, E],
        # Moving points
        0x00ff00,
        C, 'C', D, 'D',
        # end point and previous endpoints
        0xff00ff, *endpoints,
        0x0000ff,
        E, 'E',
    ]

graph = alg.graph(
    graph_func,
```

(continues on next page)

(continued from previous page)

```
    scale=0.1, animate=True,
)
# graph
ipywidgets.VBox([ac_widget, bd_widget, graph])
```

[4]:
```
VBox(children=(FloatSlider(value=4.329777002067039, max=10.0, min=-10.0, step=0.01),␣
↪FloatSlider(value=-6.4183...
```

[ ]:
```

```

5.2 3DPGA

5.2.1 3DPGA: Points and lines

Based on https://enki.ws/ganja.js/examples/coffeeshop.html#pga3d_points_and_lines.

First we create 3DPGA, and add its basis blades to the local namespace.

[1]:
```
from kingdon import Algebra

alg = Algebra(3, 0, 1)
locals().update(alg.blades)
```

Next, make formulas to construct points and planes from coefficients.

[2]:
```
# Construct points and lines from coefficients
point = lambda x, y, z: (e0 + x*e1 + y*e2 + z*e3).dual()
plane = lambda a, b, c, d: a*e1 + b*e2 + c*e3 + d*e0
```

Lets define some points to demonstrate basic incidence and rendering.

[3]:
```
A = point(0,.8,0)
B = point(.8,-1,-.8)
C = point(-.8,-1,-.8)
D = point(.8,-1,.8)
E = point(-.8,-1,.8)
```

[4]:
```
def graph_func():
    # Points can be joined into lines and planes
    ec = E & C
    p = A & B & C

    # Sum points to get the average position.
    avg = A + B + E
    bc = B + C

    # Join (&) points into lines
    l = avg & bc

    # Intersect a line and a plane into a point.
```

(continues on next page)

(continued from previous page)

```
    intersect = l ^ (A & E & D)

    # Sum lines to get average lines.
    l2 = l.normalized() + ec.normalized()

    return [
        0xD0FFE1, [A,B,D],                                          # polygons
        0x00AA88, [A,B],[A,C],[A,D],[A,E],[B,C],[C,E],[E,D],[D,B],  # edges
        0x224488, A,"A",B,"B",C,"C",D,"D",E,"E",                    # points
        0x884488, ec,"E&C", p*0.1,"A&B&C",                          # join of points
        0x884488, bc, "B+C", avg, "A+B+E", l,                       # sum of points
        0x00AA88, intersect, "line ^ plane",                        # meets
        0xFF8844, l2, "sum of lines",                               # sum of lines.
    ]

alg.graph(
    graph_func,
    lineWidth=3,
    grid=1,
    labels=1,
    h=0.6,
    p=-0.15,
    pointRadius=1,
    fontSize=1,
    scale=1,
)
```

[4]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e3', 'e01', 'e02', 'e03', 'e12', 'e13', 'e23
↪', 'e012', 'e013', 'e...
```

[ ]:
```

```

5.2.2 3DPGA: Distances and Angles

[1]:
```
from kingdon import Algebra
import math

alg = Algebra(3, 0, 1)
```

[2]:
```
point = lambda x, y, z: alg.vector(e0=1, e1=x, e2=y, e3=z).dual()
plane = lambda a, b, c, d: alg.vector(e1=a, e2=b, e3=c, e0=d)
```

Useful distances formulea:

[3]:
```
def dist_pp(P1, P2) -> float:  # point to point
    return (P1.normalized() & P2.normalized()).norm().e

def dist_pP(P, p) -> float:  # point to plane
    return (P.normalized() & p.normalized()).norm().e
```

(continues on next page)

(continued from previous page)

```
def dist_ll(l1,l2) -> float:  # line to line
    return (l1.normalized() * l2.normalized()).dual().norm().e
```

Useful angle formulea; notice that they are identical:

[4]:
```
def angle_pp(p1,p2):  # Angle between planes
    return math.acos((p1.normalized() | p2.normalized()).e) * 180 / math.pi

def angle_ll(l1,l2):  # Angle between lines
    return math.acos((l1.normalized() | l2.normalized()).e) * 180 / math.pi
```

[ ]:
```

```

[5]:
```
# Create 5 points.
A = point(0,.8,0)
B = point(.8,-1,-.8)
C = point(-.8,-1,-.8)
D = point(.8,-1,.8)
E = point(-.8,-1,.8)

# Our ground plane
a = B & C & D

# Our camera position and orientation.
camera = alg.evenmv(e=1, e13=1).normalized()
```

[6]:
```
from timeit import default_timer

def graph_func():
    time = default_timer() / 5
    A = point(0, math.sin(time*4), 0)
    # camera = (math.cos(time) + math.sin(time)*alg.blades.e13)                    #␣
↪rotate around Y
    return [
        0xD0FFE1,[A,B,C],                                                  # graph on face
        0x00AA88,[A,B],[A,C],[A,D],[B,C],[B,D],[C,E],[A,E],[E,D],          # graph all edges
        0x444444,A,"A",B,"B",C,"C",D,"D",E,"E",                            # graph all vertices
        0xFF8844,[A,E], f"{dist_pp(A,E):.2f}",                            # distance A to E
        0x224488,[A,B+E], f"{dist_pP(A,a):.2f}",                          # distance A to a
        0x44aa44,C+E, f"{dist_ll(C&E,D&B):.2f}",                          # distance CE to DB
        0x44aaff,[A+D+E,B+C+5*D+5*E,D+E], f"{angle_pp(A&E&D, a):.2f}"+"&deg;",   # angle␣
↪planes.
        0x884488,[A,2*A+D,2*A+B],f"{angle_ll(A&D, B&A):.2f}"+"&deg;"
    ]

alg.graph(
    graph_func,
    animate=True, grid=True, labels=True, lineWidth=3,
    # camera=camera,
)
```

[6]:
```
GraphWidget(cayley=[['1', 'e0', 'e1', 'e2', 'e3', 'e01', 'e02', 'e03', 'e12', 'e13', 'e23
↪', 'e012', 'e013', 'e...
```

[ ]:
```

```

5.3 3DCGA

5.3.1 3DCGA: Intersections

Based on https://enki.ws/ganja.js/examples/coffeeshop.html#cga3d_intersections.

First we create 3DCGA, and add its basis blades to the local namespace.

[1]:
```
from kingdon import Algebra

alg = Algebra(4, 1)
locals().update(alg.blades)
```

We start by defining a null basis, and upcasting for points

[2]:
```
ni = e4 + e5
no = 0.5 * (e5 - e4)
nino = ni ^ no
up = lambda x: no + x + 0.5 * x * x * ni
```

Next we'll define some objects.

[3]:
```
p  = up(0)                          # point
S  = (p - 0.5 * ni).dual()                 # main dual sphere around point (interactive)
S2 = (up(-1.4*e1) - 0.125 * ni).dual()         # left dual sphere
C  = (up(1.4*e1) - 0.125 * ni).dual() & e3.dual()    # right circle
L  = up(0.9*e2) ^ up(0.9*e2 - 1*e1) ^ ni       # top line
P  = (1*e2 - 0.9*ni).dual()                   # bottom dual plane
P2 = (1*e1 + 1.7*ni).dual()                   # right dual plane
```

[4]:
```
C1 = S & P
C2 = S & L
C3 = S & S2
C4 = S & C
C5 = C & P2
lp = up(nino.lc(P2 & (L ^ no)))
```

[5]:
```
alg.graph(
    0x00FF0000, p, "s1",             # point
    0xFF00FF,lp,"l&p",               # line intersect plane
    0x0000FF,C1,"s&p",               # sphere meet plane
    0x888800,C2,"s&l",               # sphere meet line
    0x0088FF,C3,"s&s",               # sphere meet sphere
    0x008800,C4,"s&c",               # sphere meet circle
    0x880000,C5,"c&p",               # circle meet sphere
```

(continues on next page)

(continued from previous page)

```
    0,L,0,C,                        # line and circle
    0xE0008800, P, P2,              # plane
    0xE0FFFFFF, S, "s1", S2,        # spheres
    conformal=1,
    grid=1,
    gl=1,
)
```

[5]:
```
GraphWidget(cayley=[['1', 'e1', 'e2', 'e3', 'e4', 'e5', 'e12', 'e13', 'e14', 'e15', 'e23
↪', 'e24', 'e25', 'e34'...
```

[ ]:
```

```

5.3.2 Writing high(ish) performance code with Kingdon and Numba

In this document we will demonstrate how easy it is to compose together the fundamental (binary and unary) operators that kingdon supplies out of the box, while maintaining good performance.

The example used is inspired by that of `clifford <https://clifford.readthedocs.io/en/latest/tutorials/PerformanceCliffordTutorial.html>`__ to allow for comparison of both syntax and performance. In order to make the performance comparison fair, we will run both versions within this document, such that both examples are run on the same machine.

First we will run `kingdon` with `numba` enabled and while leveraging the sparseness in the example, to get the best speeds possible. Then we will run the `clifford` equivalent. Lastly, we will run the `kingdon` example again but with a full multivector, to show how the two libraries compare in this case.

Kingdon unchained

First, we run `kingdon` on the example, allowing it to use all of its tricks.

Let's initiate a 3DCGA algebra and create some simple lines.

[1]:
```
from kingdon import Algebra, MultiVector
import numpy as np
from numba import njit

alg = Algebra(4, 1, wrapper=njit)
locals().update(alg.blades)
```

In order to have a fair comparison, we make a full multivector, which prevents `kingdon` from using the sparseness of the input to gain additional performance benefits.

[2]:
```
l1 = e145
l2 = e245
R = 1 + l2 * l1
```

The function we will optimize for this example is the conjugation (or sandwich product) used to transform a multivector X by a rotor R:

[3]:
```
def apply_rotor(R, X):
    return R * X * ~R
```

The first execution will be the most expensive, because it triggers the code generation for the geometric product and reversion operators to be performed. We therefore time it seperatelly:

[4]:
```
%timeit -n 1 -r 1 apply_rotor(R, l1)
```

```
153 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

We can now time the actual excecution time of the `apply_rotor` function.

[5]:
```
%timeit -n 100_000 -r 10 apply_rotor(R, l1)
```

```
13.4 µs ± 5.47 µs per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

We now have a benchmark: to perform the function `apply_rotor` takes about 7.69 µs ± 59.2 ns on the authors laptop. To do better is easy: we simply apply the `Algebra.register` decorator to `apply_rotor`:

```
@alg.register
def apply_rotor(R, X):
    return R * X * ~R
```

This decorator allows `kingdon` to work its magic on the decorated function. While the decorator syntax is pleasant to look it, it overwrites the original function, so in this document we are better off using

```
apply_rotor_compiled = alg.register(apply_rotor)
```

[6]:
```
apply_rotor_compiled = alg.register(apply_rotor)
```

[7]:
```
%timeit -n 1 -r 1 apply_rotor_compiled(R, l1)
```

```
104 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[8]:
```
%timeit -n 100_000 -r 10 apply_rotor_compiled(R, l1)
```

```
8.89 µs ± 975 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

After decoration the code runs in about 3.87 µs ± 529 ns on the authors laptop, which is roughly a two-fold improvement in performance. Not bad, for so little effort. The reason for the speed-up is that ordinarilly kingdon has to traverse down to the actual numerical implementation from the python level and then back up for every operator. So in this example this is done three times: two times for the products *, and once for the reversion ~.

By contrast, the `Algebra.register` decorator only traverses down to the numerical level once, does all the computations on that level, and then passes the results back up the chain.

Having had our first taste of better performance, we of course want to know if we can do even better. In this example we would expect that we can get a better result if we first do some symbolical symplification, since both R and $\widetilde{R}$ appear in the expression. We can ask the decorator to do symbolic symplification by instead using

```
@alg.register(symbolic=True)
def apply_rotor(R, X):
    return R * X * ~R
```

(Note: for more complicated expressions this might be a bad idea, since symbolic symplification can get very expensive.)

[9]:
```
apply_rotor_simplified = alg.register(apply_rotor, symbolic=True)
```

[10]:
```
%timeit -n 1 -r 1 apply_rotor_simplified(R, l1)
```

```
710 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[11]:
```
%timeit -n 100_000 -r 10 apply_rotor_simplified(R, l1)
```

```
4.17 µs ± 330 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

The symbolical optimization is more expensive to perform, even in this sparse example. However, we do obtain a new speed record: 1.91 µs ± 376 ns, which is ~4 times faster than the original.

Of course conjugation is a common operation, so `kingdon` does ship with a precompiled version. The operator for conjugation is given by >>:

[12]:
```
%timeit -n 1 -r 1 R >> l1
```

```
95.6 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[13]:
```
%timeit -n 100_000 -r 10 R >> l1
```

```
8.69 µs ± 812 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

This has comparable numerical performance and also has a shorter generation time, which is important for the user experience.

We have now achieved the limit of what can be achieved with the `@alg.register` decorator: 1.91 µs ± 376 ns, a four-fold speed-up compared to the undecorated version. Beware however, that until we can find a faster symbolic engine than `sympy`, it is usually a bad idea to use the `symbolic` keyword. It is therefore recommended to only use the `symbolic` keyword selectly.

Clifford

Now let us repeat these exercises, but using `clifford <https://clifford.readthedocs.io/en/latest/tutorials/PerformanceCliffordTutorial.html>`__.

[14]:
```
import clifford as cf
from clifford import g3c
import numba

# Get the layout in our local namespace etc etc
layout = g3c.layout
locals().update(g3c.blades)

ep, en, up, down, homo, E0, ninf, no = (g3c.stuff["ep"], g3c.stuff["en"],
                                        g3c.stuff["up"], g3c.stuff["down"], g3c.stuff[
↪"homo"],
                                        g3c.stuff["E0"], g3c.stuff["einf"], -g3c.stuff[
↪"eo"])
# Define a few useful terms
E = ninf^(no)
I5 = e12345
I3 = e123
```

[15]:
```
def cf_apply_rotor(R,mv):
    return R*mv*~R
```

[16]:
```
line_one = (up(0)^up(e1)^ninf).normal()
line_two = (up(0)^up(e2)^ninf).normal()
R = 1 + line_two*line_one
```

[17]:
```
%timeit -n 1 -r 1 cf_apply_rotor(R, line_one)
%timeit -n 100_000 -r 10 cf_apply_rotor(R, line_one)
```

```
113 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
23.3 µs ± 5.54 µs per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

The default `clifford` version of the conjugation formula takes 9.14 µs ± 55.6 ns on the authors laptop. In order to improve performance, we need to reach into the internals of `clifford`, and explicitelly call the relevant functions. This means that to speed-up the code, one has to be an advanced user.

[18]:
```
def cf_apply_rotor_faster(R,mv):
    return layout.MultiVector(layout.gmt_func(R.value,layout.gmt_func(mv.value,layout.
↪adjoint_func(R.value))) )
```

[19]:
```
%timeit -n 1 -r 1 cf_apply_rotor_faster(R, line_one)
%timeit -n 100_000 -r 10 cf_apply_rotor_faster(R, line_one)
```

```
43.6 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
6.25 µs ± 823 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

The result is very good: about a two-fold speed-up to 4.95 µs ± 698 ns on the authors laptop.

We can improve this further by adding the `numba.njit` decorator to our function:

[20]:
```
gmt_func = layout.gmt_func
adjoint_func = layout.adjoint_func

@numba.njit
def cf_apply_rotor_val_numba(R_val,mv_val):
    return gmt_func(R_val, gmt_func(mv_val, adjoint_func(R_val)))

def cf_apply_rotor_wrapped_numba(R,mv):
    return cf.MultiVector(layout, cf_apply_rotor_val_numba(R.value, mv.value))
```

[21]:
```
%timeit -n 1 -r 1 cf_apply_rotor_wrapped_numba(R, line_one)
%timeit -n 100_000 -r 10 cf_apply_rotor_wrapped_numba(R, line_one)
```

```
90.2 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
4.12 µs ± 326 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

We are now down to 3.52 µs ± 403 ns per loop on the authors laptop, a ~2.6 times increase. These times however are very comparible to the 3.87 µs ± 529 ns we got from the `@alg.register` decorator, which looked like this:

```
@alg.register
def apply_rotor(R, X):
    return R * X * ~R
```

So with a much cleaner API, we can achieve similar results in `kingdon`.

King on a Leash

In order to have a fair comparison, we will end with a computation in kingdon on a full multivector. This means `kingdon` will not be able to cheat by doing almost no computations in the first place, and will really have to multiply full multivectors, just like `clifford`.

[22]:
```
from kingdon import Algebra, MultiVector
import numpy as np

alg = Algebra(4, 1, wrapper=njit)
locals().update(alg.blades)
```

[23]:
```
l1vals = np.zeros(len(alg))
l2vals = np.zeros(len(alg))
l1 = alg.multivector(l1vals) + e145
l2 = alg.multivector(l2vals) + e245
R = 1 + l2 * l1
```

[24]:
```
def apply_rotor(R, X):
    return R * X * ~R
```

[25]:
```
%timeit -n 1 -r 1 apply_rotor(R, l1)
%timeit -n 100_000 -r 10 apply_rotor(R, l1)
```

```
254 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
20.3 µs ± 507 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

On a full multivector, `kingdon` takes 18 µs ± 696 ns per loop on the authors laptop. This is significantly more than the 7.69 µs ± 59.2 ns for the sparse scenario, and also more than the 9.14 µs ± 55.6 ns delivered by `clifford`. However, we can fix this in one simple move:

[26]:
```
apply_rotor_compiled = alg.register(apply_rotor)
```

[27]:
```
%timeit -n 1 -r 1 apply_rotor_compiled(R, l1)
%timeit -n 100_000 -r 10 apply_rotor_compiled(R, l1)
```

```
181 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
5.28 µs ± 100 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

With one decorator, we are down to 5.59 µs ± 724 ns per loop, which is close to the 3.52 µs ± 403 ns that `clifford` achieves with much more manual labour on the part of the programmer. (We will not include the `symbolic=True` version here, because for a full multivector this will be way to expensive.)

Most of the costs that both `clifford` and `kingdon` make is in the glue around the computation. This is responsable for part of the speed-up in the `clifford` code, and we haven't tapped into this yet in the `kingdon` example so far. So if you are willing to compromise readability, `kingdon` can still do a little bit better by removing the glue:

[28]:
```
def apply_rotor_numba(R, X):
    keys_out, func = apply_rotor_compiled[R.keys(), X.keys()]
    numba_func = alg.numspace[func.__name__]  # The numba version is available from the␣
↪namespace of numerical functions; numspace.
    return MultiVector.fromkeysvalues(
        alg,
        keys=keys_out,
```

(continues on next page)

(continued from previous page)

```
        values=numba_func(R.values(), X.values()),
    )
```

The decorated function `apply_rotor_compiled` can be accessed like a dictionary to retrieve the numerical function and the resulting `keys_out` for the input `keys` of `R` and `l1`. Moreover, the alternative constructor `MultiVector.fromkeysvalues` bypasses the sanity checking of the default constructor so as to reach optimal performance. We then simply have to make a function to return the result back into

[29]:
```
%timeit -n 1 -r 1 apply_rotor_numba(R, l1)
```

```
27.5 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[30]:
```
%timeit -n 100_000 -r 10 apply_rotor_numba(R, l1)
```

```
3.05 µs ± 61.2 ns per loop (mean ± std. dev. of 10 runs, 100,000 loops each)
```

We now see that the `kingdon` function is also faster than the `clifford` version in the full multivector version, with 2.91 µs ± 16.3 ns for `kingdon` vs. 3.52 µs ± 403 ns for `clifford`. Putting the `kingdon` decorator on a diet to reach this limit is an ongoing effort.

Conclusion

For full multivectors, clifford is still a bit faster than `kingdon`'s `Algebra.register` decorator, but at the cost of readability. And as we have seen, `kingdon` can match this performance and lack of readability if needed. However, in reality many multivectors are not full, but rather of only a select number of grades, or even just a couple of coefficients. In these scenarios, `kingdon` really comes into its own, and the very readable `Algebra.register` decorator offers a way to achieve high performance for very little effort on the users part.

[ ]:
```
```

CHAPTER

SIX

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/tbuli/kingdon/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

Kingdon could always use more documentation, whether as part of the official Kingdon docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/tbuli/kingdon/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *kingdon* for local development.

1. Fork the *kingdon* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/kingdon.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv kingdon
$ cd kingdon/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 kingdon tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.8, 3.9, and 3.10, and for PyPy. Check https://travis-ci.com/tbuli/kingdon/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ pytest tests.test_kingdon
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push origin <tag_name>
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SEVEN

CREDITS

This package was inspired by GAmphetamine.js.

7.1 Development Lead

- Martin Roelfs <martinroelfs@yahoo.com>

7.2 Contributors

None yet. Why not be the first?

CHAPTER

EIGHT

MODULE DOCUMENTATION

8.1 Algebra

class kingdon.algebra.**Algebra**(*p: int = 0, q: int = 0, r: int = 0, signature: ~numpy.ndarray = None, start_index: int = None, registry: dict = <factory>, numspace: dict = <factory>, cse: bool = True, graded: bool = False, wrapper: ~collections.abc.Callable = None, codegen_symbolcls: object = <bound method RationalPolynomial.fromname of <class 'kingdon.polynomial.RationalPolynomial'>>, simp_func: ~collections.abc.Callable = <function Algebra.<lambda>>*)

A Geometric (Clifford) algebra with `p` positive dimensions, `q` negative dimensions, and `r` null dimensions.

The default settings of `cse = simplify = True` usually strike a good balance between initiation times and subsequent code execution times.

Parameters

- **p** – number of positive dimensions.
- **q** – number of negative dimensions.
- **r** – number of null dimensions.
- **cse** – If `True` (default), attempt Common Subexpression Elimination (CSE) on symbolically optimized expressions.
- **graded** – If `True` (default is `False`), perform binary and unary operations on a graded basis. This will still be more sparse than computing with a full multivector, but not as sparse as possible. It does however, vastly reduce the number of possible expressions that have to be symbolically optimized.
- **simplify** – If `True` (default), we attempt to simplify as much as possible. Setting this to `False` will reduce the number of calls to simplify. However, it seems that `True` is still faster, probably because it keeps sympy expressions from growing too large, which makes both symbolic computations and printing into a python function slower.
- **wrapper** – A function that is always applied to the generated functions as a decorator. For example, using `numba.njit` as a wrapper will ensure that all kingdon code is jitted using numba.
- **codegen_symbolcls** – The symbol class used during codegen. By default, this is our own fast *`RationalPolynomial`* class.
- **simp_func** – This function is applied as a filter function to every multivector coefficient.

acp: *`OperatorDict`*

add: *OperatorDict*

bin2canon: dict

bivector(**args*, ***kwargs*) → *MultiVector*

blades: *BladeDict*

canon2bin: dict

cayley: dict

classmethod codegen_symbolcls(*name*)

conjugate: *UnaryOperatorDict*

cp: *OperatorDict*

cse: bool = True

d: int

div: *OperatorDict*

evenmv(**args*, ***kwargs*) → *MultiVector*

Create a new *MultiVector* in the even subalgebra.

property frame: list

The set of orthogonal basis vectors, $\{e_i\}$. Note that for a frame linear independence suffices, but we already have orthogonal basis vectors so why not use those?

gp: *OperatorDict*

graded: bool = False

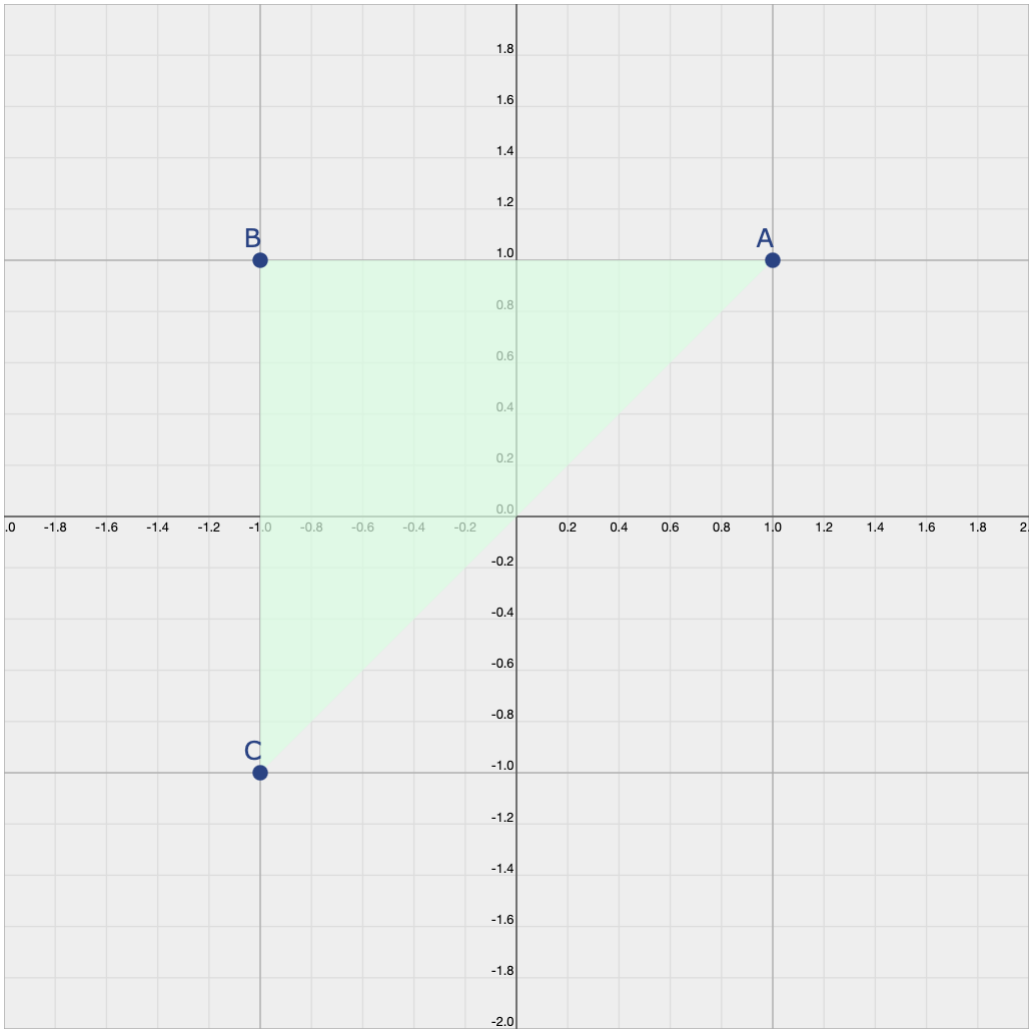
graph(**subjects*, *graph_widget=<class 'kingdon.graph.GraphWidget'>*, ***options*)

The graph function outputs `ganja.js` renders and is meant for use in jupyter notebooks. The syntax of the graph function will feel familiar to users of `ganja.js`: all position arguments are considered as subjects to graph, while all keyword arguments are interpreted as options to `ganja.js`'s `Algebra.graph` method.

Example usage:

```
alg.graph(
    0xD0FFE1, [A,B,C],
    0x224488, A, "A", B, "B", C, "C",
    lineWidth=3, grid=1, labels=1
)
```

Will create



If a function is given to `Algebra.graph` then it is called without arguments. This can be used to make animations in a manner identical to `ganja.js`.

Example usage:

```
def graph_func():
    return [
        0xD0FFE1, [A,B,C],
        0x224488, A, "A", B, "B", C, "C"
    ]

alg.graph(
    graph_func,
    lineWidth=3, grid=1, labels=1
)
```

Parameters

- ***subjects** – Subjects to be graphed. Can be strings, hexadecimal colors, (lists of) MultiVector, (lists of) callables.
- ****options** – Options passed to `ganja.js`'s `Algebra.graph`.

hodge: *UnaryOperatorDict*

property indices_for_grade

Mapping from the grades to the indices for that grade. E.g. in 2D VGA, this returns

```
{0: (0,), 1: (1, 2), 2: (3,)}
```

property indices_for_grades

Mapping from a sequence of grades to the corresponding indices. E.g. in 2D VGA, this returns

```
{(): (), (0,): (0,), (1,): (1, 2), (2,): (3,), (0, 1): (0, 1, 2),
 (0, 2): (0, 3), (1, 2): (1, 2, 3), (0, 1, 2): (0, 1, 2, 3)}
```

inv: *UnaryOperatorDict*

involute: *UnaryOperatorDict*

ip: *OperatorDict*

lc: *OperatorDict*

property matrix_basis

multivector(**args*, ***kwargs*) → *MultiVector*

Create a new *MultiVector*.

neg: *UnaryOperatorDict*

normsq: *UnaryOperatorDict*

numspace: dict

oddmv(**args*, ***kwargs*) → *MultiVector*

Create a new *MultiVector* of odd grades. (There is technically no such thing as an odd subalgebra, but otherwise this is similar to *evenmv*.)

op: *OperatorDict*

outercos: *UnaryOperatorDict*

outerexp: *UnaryOperatorDict*

outersin: *UnaryOperatorDict*

outertan: *UnaryOperatorDict*

p: int = 0

polarity: *UnaryOperatorDict*

proj: *Registry*

pseudobivector(**args*, ***kwargs*) → *MultiVector*

pseudoquadvector(**args*, ***kwargs*) → *MultiVector*

pseudoscalar(**args*, ***kwargs*) → *MultiVector*

pseudotrivector(**args*, ***kwargs*) → *MultiVector*

pseudovector(**args*, ***kwargs*) → *MultiVector*

pss: object

purevector(**args*, *grade*, ***kwargs*) → *MultiVector*

Create a new `MultiVector` of a specific grade.

Parameters

grade – Grade of the mutivector to create.

q: int = 0

quadvector(**args*, ***kwargs*) → *MultiVector*

r: int = 0

rc: *OperatorDict*

property reciprocal_frame: list

The reciprocal frame is a set of vectors $\{e^i\}$ that satisfies $e^i \cdot e_j = \delta^i_j$ with the frame vectors $\{e_i\}$.

register(*expr=None*, */*, ***, *name=None*, *symbolic=False*)

Register a function with the algebra to optimize its execution times.

The function must be a valid GA expression, not an arbitrary python function.

Example:

```
@alg.register
def myexpr(a, b):
    return a @ b

@alg.register(symbolic=True)
def myexpr(a, b):
    return a @ b
```

With default settings, the decorator will ensure that every GA unary or binary operator is replaced by the corresponding numerical function, and produces numerically much more performant code. The speed up is particularly notible when e.g. *self.wrapper=numba.njit*, because then the cost for all the python glue surrounding the actual computation has to be paid only once.

When *symbolic=True* the expression is symbolically optimized before being turned into a numerical function. Beware that symbolic optimization of longer expressions (currently) takes exorbitant amounts of time, and often isn't worth it if the end goal is numerical computation.

Parameters

- **expr** – Python function of a valid kingdon GA expression.
- **name** – (optional) name by which the function will be known to the algebra. By default, this is the *expr.__name__*.
- **symbolic** – (optional) If true, the expression is symbolically optimized. By default this is False, given the cost of optimizing large expressions.

registry: dict

reverse: *UnaryOperatorDict*

rp: *OperatorDict*

scalar(**args*, ***kwargs*) → *MultiVector*

signature: ndarray = None

signs: dict

simp_func()

sp: *OperatorDict*

sqrt: *UnaryOperatorDict*

start_index: int = None

sub: *OperatorDict*

sw: *Registry*

trivector(**args*, ***kwargs*) → *MultiVector*

unhodge: *UnaryOperatorDict*

unpolarity: *UnaryOperatorDict*

vector(**args*, ***kwargs*) → *MultiVector*

wrapper: Callable = None

class kingdon.algebra.**BladeDict**(*algebra:* Algebra, *lazy: bool = False*)

Dictionary of basis blades. Use getitem or getattr to retrieve a basis blade from this dict, e.g.:

```
alg = Algebra(3, 0, 1)
blade_dict = BladeDict(alg, lazy=True)
blade_dict['e03']
blade_dict.e03
```

When *lazy=True*, the basis blade is only initiated when requested. This is done for performance in higher dimensional algebras.

algebra: *Algebra*

blades: dict

lazy: bool = False

8.2 MultiVector

class kingdon.multivector.**MultiVector**(*algebra: 'Algebra', values=None, keys=None, *, name=None, grades=None, symbolcls=<class 'sympy.core.symbol.Symbol'>, **items*)

acp(*other*)

Calculate the anti-commutator product of `x := self` and `y := other`: `x.cp(y) = 0.5*(x*y+y*x)`.

add(*other*)

algebra: *Algebra*

asfullmv(*canonical=True*)

Returns a full version of the same multivector.

Parameters
canonical – If True (default) the values are in canonical order, even if the mutivector was already dense.

asmatrix()

Returns a matrix representation of this multivector.

conjugate()

Clifford conjugation: involution and reversion combined.

cp(*other*)

Calculate the commutator product of `x := self` and `y := other`: `x.cp(y) = 0.5*(x*y-y*x)`.

div(*other*)

dual(*kind='auto'*)

Compute the dual of *self*. There are three different kinds of duality in common usage. The first is polarity, which is simply multiplying by the inverse PSS. This is the only game in town for non-degenerate metrics (Algebra.r = 0). However, for degenerate spaces this no longer works, and we have two popular options: Poincaré and Hodge duality.

By default, `kingdon` will use polarity in non-degenerate spaces, and Hodge duality for spaces with *Algebra.r = 1*. For spaces with *r > 2*, little to no literature exists, and you are on your own.

Parameters
kind – if 'auto' (default), `kingdon` will try to determine the best dual on the basis of the signature of the space. See explenation above. To ensure polarity, use `kind='polarity'`, and to ensure Hodge duality, use `kind='hodge'`.

filter(*func=None*) → *MultiVector*

Returns a new multivector containing only those elements for which *func* was true-ish. If no function was provided, use the simp_func of the Algebra.

property free_symbols

classmethod fromkeysvalues(*algebra*, *keys*, *values*)

Initiate a multivector from a sequence of keys and a sequence of values.

classmethod frommatrix(*algebra*, *matrix*)

Initiate a multivector from a matrix. This matrix is assumed to be generated by `asmatrix`, and thus we only read the first column of the input matrix.

gp(*other*)

grade(**grades*)

Returns a new `MultiVector` instance with only the selected *grades* from *self*.

Parameters
grades – tuple or ints, grades to select.

property grades

Tuple of the grades present in *self*.

hodge()

inv()

Inverse of this multivector.

involute()

Main grade involution.

ip(*other*)

property issymbolic

True if this mv contains Symbols, False otherwise.

items()

itermv(*axis=None*) → Generator[*MultiVector*, None, None]

Returns an iterator over the multivectors within this multivector, if it is a multidimensional multivector. For example, if you have a pointcloud of N points, itermv will iterate over these points one at a time.

Parameters

axis – Axis over which to iterate. Default is to iterate over all possible mv.

keys()

lc(*other*)

map(*func*) → *MultiVector*

Returns a new multivector where *func* has been applied to all the values.

neg()

norm()

normalized()

Normalized version of this multivector.

normsq()

op(*other*)

outercos()

outerexp()

outersin()

outertan()

polarity()

proj(*other*)

Project `x := self` onto `y := other`: `x @ y = (x | y) * ~y`. For correct behavior, `x` and `y` should be normalized (k-reflections).

rc(*other*)

reverse()

Reversion

rp(*other*)

property shape

Return the shape of the .values() attribute of this multivector.

sp(*other*)

Scalar product: $\langle x \cdot yangle$.

sqrt()

sub(*other*)

sw(*other*)

Apply `x := self` to `y := other` under conjugation: `x.sw(y) = x*y*~x`.

property type_number: int

undual(*kind='auto'*)

Compute the undual of *self*. See `dual` for more information.

unhodge()

unpolarity()

values()

8.3 Codegen

The codegen module generates python functions from operations between/on purely symbolic `MultiVector` objects.

As a general rule, these functions take in pure symbolic `MultiVector` objects and return a tuple of keys present in the output, and a pure python function which represents the respective operation.

E.g. `codegen_gp()` computes the geometric product between two multivectors for the specific non-zero basis blades present in the input.

class kingdon.codegen.**AdditionChains**(*limit: 'int'*)

limit: int

property minimal_chains: Dict[int, Tuple[int, ...]]

class kingdon.codegen.**CodegenOutput**(*keys_out: Tuple[int], func: Callable*)

Output of a codegen function.

Parameters

- **keys_out** – tuple with the output blades in binary rep.
- **func** – callable that takes (several) sequence(s) of values returns a tuple of `len(keys_out)`.

func: Callable

Alias for field number 1

keys_out: Tuple[int]

Alias for field number 0

class kingdon.codegen.**Fraction**(*numer, denom*)

Tuple representing a fraction.

denom

Alias for field number 1

numer

Alias for field number 0

class kingdon.codegen.**KingdonPrinter**(*printer=None*, *dummify=False*)

doprint(*funcname*, *args*, *names*, *expr*, **, cses=()*)

Returns the function definition code as a string.

class kingdon.codegen.**LambdifyInput**(*funcname: str*, *args: dict*, *expr_dict: dict*, *dependencies: list*)

Strike package for the Lambdify function.

args: dict

Alias for field number 1

dependencies: list

Alias for field number 3

expr_dict: dict

Alias for field number 2

funcname: str

Alias for field number 0

class kingdon.codegen.**TermTuple**(*key_out: int*, *keys_in: Tuple[int]*, *sign: int*, *values_in: Tuple['sympy.core.symbol.Symbol']*, *termstr: str*)

TermTuple represents a single monomial in a product of multivectors.

Parameters

- **key_out** – is the basis blade to which this monomial belongs.
- **keys_in** – are the input basis blades in this monomial.
- **sign** – Sign of the monomial.
- **values_in** – Input values. Typically, tuple of `sympy.core.symbol.Symbol`.
- **termstr** – The string representation of this monomial, e.g. '-x*y'.

key_out: int

Alias for field number 0

keys_in: Tuple[int]

Alias for field number 1

sign: int

Alias for field number 2

termstr: str

Alias for field number 4

values_in: Tuple['sympy.core.symbol.Symbol']

Alias for field number 3

kingdon.codegen.**codegen_acp**(*x*, *y*)

Generate the anti-commutator product of `x` and `y`: `x.acp(y) = 0.5*(x*y+y*x)`.

Returns
tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_add(x, y)`

`kingdon.codegen.codegen_conjugate(x)`

`kingdon.codegen.codegen_cp(x, y)`

Generate the commutator product of `x` and `y`: `x.cp(y) = 0.5*(x*y-y*x)`.

Returns
tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_div(x, y)`

Generate code for xy^{-1}.

`kingdon.codegen.codegen_gp(x, y)`

Generate the geometric product between `x` and `y`.

Parameters

- **x** – Fully symbolic *MultiVector*.
- **y** – Fully symbolic *MultiVector*.

Returns
tuple with integers indicating the basis blades present in the product in binary convention, and a lambda function that perform the product.

`kingdon.codegen.codegen_hitzer_inv(x, symbolic=False)`

Generate code for the inverse of `x` using the Hitzer inverse, which works up to 5D algebras.

`kingdon.codegen.codegen_hodge(x, undual=False)`

`kingdon.codegen.codegen_inv(y, x=None, symbolic=False)`

`kingdon.codegen.codegen_involute(x)`

`kingdon.codegen.codegen_involutions(x, invert_grades=(2, 3))`

Codegen for the involutions of Clifford algebras: reverse, grade involute, and Clifford involution.

Parameters
invert_grades – The grades that flip sign under this involution mod 4, e.g. (2, 3) for reversion.

`kingdon.codegen.codegen_ip(x, y, diff_func=<built-in function abs>)`

Generate the inner product of `x` and `y`.

Parameters
diff_func – How to treat the difference between the binary reps of the basis blades. if `abs`, compute the symmetric inner product. When `lambda x: -x` this function generates left-contraction, and when `lambda x: x`, right-contraction.

Returns
tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_lc(x, y)`

Generate the left-contraction of `x` and `y`.

Returns
tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_neg(x)`

`kingdon.codegen.codegen_normsq(x)`

`kingdon.codegen.codegen_op(x, y)`

Generate the outer product of `x` and `y`: `x.op(y) = x ^ y`.

X
MultiVector

Y
MultiVector

Returns
dictionary with integer keys indicating the corresponding basis blade in binary convention, and values which are a 3-tuple of indices in x, indices in y, and a lambda function.

`kingdon.codegen.codegen_outercos(x)`

`kingdon.codegen.codegen_outerexp(x, asterms=False)`

`kingdon.codegen.codegen_outersin(x)`

`kingdon.codegen.codegen_outertan(x)`

`kingdon.codegen.codegen_polarity(x, undual=False)`

`kingdon.codegen.codegen_product(x, y, filter_func=None, sign_func=None, keyout_func=<built-in function xor>)`

Helper function for the codegen of all product-type functions.

Parameters

- **x** – Fully symbolic *MultiVector*.
- **y** – Fully symbolic *MultiVector*.
- **filter_func** – A condition which should be true in the preprocessing of terms. Input is a TermTuple.
- **sign_func** – function to compute sign between terms. E.g. algebra.signs[ei, ej] for metric dependent products. Input: 2-tuple of blade indices, e.g. (ei, ej).
- **keyout_func**

`kingdon.codegen.codegen_proj(x, y)`

Generate the projection of `x` onto `y`: $(x \cdot y)\widetilde{y}$.

Returns
tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_rc(x, y)`

Generate the right-contraction of `x` and `y`.

Returns
tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_reverse(x)`

`kingdon.codegen.codegen_rp(x, y)`

Generate the regressive product of `x` and `y`:, $x \vee y$.

Parameters

- `x`
- `y`

Returns

tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_shirokov_inv(x, symbolic=False)`

Generate code for the inverse of `x` using the Shirokov inverse, which is works in any algebra, but it can be expensive to compute.

`kingdon.codegen.codegen_sp(x, y)`

Generate the scalar product of `x` and `y`.

Returns

tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_sqrt(x)`

Take the square root using the study number approach as described in https://doi.org/10.1002/mma.8639

`kingdon.codegen.codegen_sub(x, y)`

`kingdon.codegen.codegen_sw(x, y)`

Generate the conjugation of `y` by `x`: $xy\widetilde{x}$.

Returns

tuple of keys in binary representation and a lambda function.

`kingdon.codegen.codegen_unhodge(x)`

`kingdon.codegen.codegen_unpolarity(x)`

`kingdon.codegen.do_codegen(codegen, *mvs) → CodegenOutput`

Parameters

- **codegen** – callable that performs codegen for the given `mvs`. This can be any callable that returns either a `MultiVector`, a dictionary, or an instance of `CodegenOutput`.
- **mvs** – Any remaining positional arguments are taken to be symbolic `MultiVector`'s.

Returns

Instance of `CodegenOutput`.

`kingdon.codegen.do_compile(codegen, *tapes)`

`kingdon.codegen.func_builder(res_vals: defaultdict, *mvs, funcname: str) → CodegenOutput`

Build a Python function for the product between given multivectors.

Parameters

- **res_vals** – Dict to be converted into a function. The keys correspond to the basis blades in binary, while the values are strings to be converted into source code.
- **mvs** – all the multivectors that the resulting function is a product of.
- **funcname** – Name of the function. Be aware: if a function by that name already existed, it will be overwritten.

Returns
tuple of output keys of the callable, and the callable.

`kingdon.codegen.lambdify`(*args: dict, exprs: list, funcname: str, dependencies: tuple = None, printer=<class 'sympy.printing.lambdarepr.LambdaPrinter'>, dummify=False, cse=False*)

Function that turns symbolic expressions into Python functions. Heavily inspired by `sympy`'s function by the same name, but adapted for the needs of `kingdon`.

Particularly, this version gives us more control over the names of the function and its arguments, and is more performant, particularly when the given expressions are strings.

Example usage:

```
alg = Algebra(2)
a = alg.multivector(name='a')
b = alg.multivector(name='b')
args = {'A': a.values(), 'B': b.values()}
exprs = tuple(codegen_cp(a, b).values())
func = lambdify(args, exprs, funcname='cp', cse=False)
```

This will produce the following code:

```
def cp(A, B):
    [a, a1, a2, a12] = A
    [b, b1, b2, b12] = B
    return (+a1*b2-a2*b1,)
```

It is recommended not to call this function directly, but rather to use *`do_codegen()`* which provides a clean API around this function.

Parameters

- **`args`** – dictionary of type dict[str | Symbol, tuple[Symbol]].
- **`exprs`** – tuple[Expr]
- **`funcname`** – string to be used as the bases for the name of the function.
- **`dependencies`** – These are extra expressions that can be provided such that quantities can be precomputed. For example, in the inverse of a multivector, this is used to compute the scalar denominator only once, after which all values in expr are multiplied by it. When `cse = True`, these dependencies are also included in the CSE process.
- **`cse`** – If `True` (default), CSE is applied to the expressions and dependencies. This typically greatly improves performance and reduces numba's initialization time.

Returns
Function that represents that can be used to calculate the values of exprs.

`kingdon.codegen.power_supply`(*x: MultiVector, exponents: Tuple[int, ...], operation: Callable[['MultiVector', 'MultiVector'], 'MultiVector'] = <built-in function mul>*)

Generates powers of a given multivector using the least amount of multiplications. For example, to raise a multivector x to the power $a = 15$, only 5 multiplications are needed since $x^2 = x*x$, $x^3 = x*x^2$, $x^5 = x^2*x^3$, $x^{10} = x^5 * x^5$, $x^{15} = x^5 * x^{10}$. The *`power_supply`* uses *`AdditionChains`* to determine these shortest chains.

When called with only a single integer, e.g. `power_supply(x, 15)`, iterating over it yields the above sequence in order; ending with x^{15}.

When called with a sequence of integers, the generator instead returns only the requested terms.

Parameters

- **x** – The MultiVector to be raised to a power.
- **exponents** – When an `int`, this generates the shortest possible way to get to x^a, where x

`kingdon.codegen.term_tuple(items, sign_func, keyout_func=<built-in function xor>)`

Create a single term in a multivector product between the basis blades present in *items*.

8.4 Operator dicts

exception `kingdon.operator_dict.AlgebraError`

class `kingdon.operator_dict.OperatorDict(name: str, codegen: Callable, algebra: Algebra)`

A dict-like object which performs codegen of a particular operator, and caches the result for future use. For example, to generate the geometric product, we create an OperatorDict as follows:

```
alg = Algebra(3, 0, 1)
gp = OperatorDict('gp', codegen=codegen_gp, algebra=alg)
```

Here, `codegen_gp` is a function that outputs the keys of the result, and a callable that produces the corresponding values. See `CodegenOutput` for more info.

algebra: *Algebra*

codegen: Callable

filter(*keys_out*, *values_out*)

For given keys and values, keep only symbolically non-zero elements.

name: str

operator_dict: dict

class `kingdon.operator_dict.Registry(name: str, codegen: Callable, algebra: Algebra)`

class `kingdon.operator_dict.UnaryOperatorDict(name: str, codegen: Callable, algebra: Algebra)`

Specialization of OperatorDict for unary operators. In the case of unary operators, we can do away with all of the overhead that is necessary for operators that act on multiple multivectors.

8.5 Matrix reps

This module contains support functions to turn `MultiVector`'s into matrices.

This follows the approach outlined in Graded Symmetry Groups: Plane and Simple, section 10. See the paper for more details.

`kingdon.matrixreps.expr_as_matrix(expr: Callable, *inputs, res_like: MultiVector = None)`

This represents any GA expression as a matrix. To illustrate by example, we might want to represent the multivector equation y = R >> x as a matrix equation y = Ax. To obtain A, call this function as follows:

```
alg = Algebra(3, 0, 1)
R = alg.evenmv(name='R')
x = alg.vector(name='x')
A, y = expr_as_matrix(lambda R, x: R >> x, R, x)
```

In order to build the matrix rep the input *expr* is evaluated, so make sure the inputs to the expression are given in the correct order. The last of the positional arguments is assumed to be the vector x.

Expr
Callable representing a valid GA expression. Can also be a `OperatorDict`.

Inputs
All positional arguments are consider symbolic input arguments to *expr*. The last of these is assumed to represent the vector x in $y = Ax$.

Res_like
optional multivector corresponding to the desired output. If None, then the full output is returned. However, if only a subsegment of the output is desired, provide a multivector with the desired shape. In the example above setting, *res_like = alg.vector(e1=1)* would mean only the e1 component of the matrix is returned. This does not have to be a symbolic multivector, only the keys are checked.

Returns
This function returns the matrix representation, and the result of applying the expression to the input.

`kingdon.matrixreps.`**`matrix_rep`**(*p=0*, *q=0*, *r=0*)

Create the matrix reps of all the basis blades of an algebra. These are selected such that the entries in the first column of the matrix have positive sign, and thus matrix-matrix multiplication is identical to matrix-vector multiplication.

Parameters

- **p** – number of positive dimensions.
- **q** – number of negative dimensions.
- **r** – number of null dimensions.

Returns
sequence of matrix reps for the basis-blades.

`kingdon.matrixreps.`**`ordering_matrix`**(*Rs*)

Matrix reps are determined up to similarity transform. But not all similarity transforms are equal. This function creates the one similarity transform that gives all entries in the first column of the matrix a positive sign. In doing so, matrix-matrix multiplication is identical to matrix-vector multiplication.

Parameters
Rs – sequence of matrix reps for all the basis blades of an algebra.

Returns
The similarity transform to beat all similarity transforms.

8.6 Graph

class `kingdon.graph.`**`GraphWidget`**(***kwargs: Any*)

algebra

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

cayley

An instance of a Python list.

draggable_points

An instance of a Python list.

draggable_points_idxs

An instance of a Python list.

get_cayley

get_draggable_points

get_draggable_points_idxs

get_key2idx

get_pre_subjects

get_signature

get_subjects

inplacereplace(*old_subjects*, *new_subjects: List[Tuple[int, dict]]*)

Given the old and the new subjects, replace the values inplace iff they have changed.

key2idx

An instance of a Python dict.

One or more traits can be passed to the constructor to validate the keys and/or values of the dict. If you need more detailed validation, you may use a custom validator method.

Changed in version 5.0: Added key_trait for validating dict keys.

Changed in version 5.0: Deprecated ambiguous `trait`, `traits` args in favor of `value_trait`, `per_key_traits`.

options

An instance of a Python dict.

One or more traits can be passed to the constructor to validate the keys and/or values of the dict. If you need more detailed validation, you may use a custom validator method.

Changed in version 5.0: Added key_trait for validating dict keys.

Changed in version 5.0: Deprecated ambiguous `trait`, `traits` args in favor of `value_trait`, `per_key_traits`.

pre_subjects

An instance of a Python list.

raw_subjects

An instance of a Python list.

signature

An instance of a Python list.

subjects

An instance of a Python list.

`kingdon.graph.`**encode**(*o*, *tree_types=(<class 'list'>, <class 'tuple'>)*, *root=False*)

kingdon.graph.**walker**(*encoded_generator*, *tree_types=(<class 'list'>, <class 'tuple'>)*)

8.7 Rational Polynomial

class kingdon.polynomial.**Polynomial**(*coeff*)

args: List[list]

classmethod fromname(*name*)

tosympy()

Return a sympy version of this Polynomial.

class kingdon.polynomial.**RationalPolynomial**(*numer*, *denom=None*)

denom: *Polynomial*

classmethod fromname(*name*)

inv()

numer: *Polynomial*

tosympy()

Return a sympy version of this Polynomial.

kingdon.polynomial.**compare**(*a*, *b*)

CHAPTER

NINE

HISTORY

9.1 0.1.0 (2023-08-12)

- First release on PyPI.

9.2 0.2.0 (2024-01-09)

- Multivectors now have *map* and *filter* methods to apply element-wise operations to the coefficients.
- Make matrix representations of expressions using *expr_as_matrix*.
- Bugfixes.

9.3 0.3.0 (2024-03-11)

- Much faster codegen by the introduction of a GAmphetamine.js inspired RationalPolynomial class, which now replaces SymPy for codegen. Particularly for inverses this is orders of magnitude faster.
- Performed a numbotomy: numba is no longer a dependency since it actually didn't add much in most cases. Instead the user can now provide the Algebra with any wrapper function, which is applied to the generated functions. This can be numba.njit, but also any other decorator.

9.4 0.3.2 (2024-03-18)

- Fixed a high priority bug in the graph function.
- Fixed a bug that stopped multivectors from being callable.

9.5 1.0.0 (2024-04-17)

- Kingdon now has proper support for ganja.js animations and the graphs are interactive!
- Indexing a multivector will no longer access coefficients. The whole promise of GA is coordinate independence, so why would you need to access coefficients? Instead, slicing a multivector will pass on that information to the underlying datastructures (e.g. numpy array or pytorch tensor), and will return a new multivector. Moreover, you can use the new slicing syntax to set values as well. If you really still need access to the coefficients, there is always the getattr syntax or the .values() method.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

E

F

G

H

I

K

S

T

U

V

W