Clifford Documentation

Release 1.3.1

Clifford Team

Jun 03, 2020

CONTENTS

```
In [1]: from clifford.g3 import *  # import GA for 3D space

In [2]: import math

In [3]: a = e1 + 2*e2 + 3*e3  # vector

In [4]: R = math.e**(math.pi/4*e12)  # rotor

In [5]: R*a*~R    # rotate the vector
Out[5]: (2.0^e1) - (1.0^e2) + (3.0^e3)
```

This module implements Geometric Algebras (a.k.a. Clifford algebras). Geometric Algebra (GA) is a universal algebra which subsumes complex algebra, quaternions, linear algebra and several other independent mathematical systems. Scalars, vectors, and higher-grade entities can be mixed freely and consistently in the form of mixed-grade multivectors.

CHAPTER

ONE

API

1.1 clifford (`clifford`)

The top-level module. Provides two core classes, `Layout` and `MultiVector`, along with several helper functions to implement the algebras.

1.1.1 Constructing algebras

Note that typically the predefined-algebras are sufficient, and there is no need to build an algebra from scratch.

`Cl`([p, q, r, sig, names, firstIdx, mvClass])	Returns a `Layout` and basis blade `MultiVector`s for the geometric algebra $Cl_{p,q,r}$.
`conformalize`(layout[, added_sig, mvClass])	Conformalize a Geometric Algebra

clifford.Cl

clifford.**Cl**(*p=0*, *q=0*, *r=0*, *sig=None*, *names=None*, *firstIdx=1*, *mvClass=<class 'clifford._multivector.MultiVector'>*)

Returns a `Layout` and basis blade `MultiVector`s for the geometric algebra $Cl_{p,q,r}$.

The notation $Cl_{p,q,r}$ means that the algebra is $p+q+r$-dimensional, with the first p vectors with positive signature, the next q vectors negative, and the final r vectors with null signature.

Returns

- **layout** (*Layout*) – The resulting layout
- **blades** (*Dict[str, MultiVector]*) – The blades of the returned layout, equivalent to `layout.blades`.

clifford.conformalize

clifford.**conformalize**(*layout*, *added_sig=[1, -1]*, ***, *mvClass=<class 'clifford._multivector.MultiVector'>*, ***kwargs*)

Conformalize a Geometric Algebra

Given the *Layout* for a GA of signature (p, q), this will produce a GA of signature (p+1, q+1), as well as return a new list of blades and some *stuff*. *stuff* is a dict containing the null basis blades, and some up/down functions for projecting in/out of the CGA.

Parameters

- **layout** (*clifford.Layout*) – layout of the GA to conformalize (the base)
- **added_sig** (*list-like*) – list of +1, -1 denoted the added signatures
- ****kwargs** – passed to Cl() used to generate conformal layout

Returns

- **layout_c** (*ConformalLayout*) – layout of the base GA
- **blades_c** (*dict*) – blades for the CGA
- **stuff** (*dict*) – dict mapping the following members of *ConformalLayout* by their names, for easy unpacking into the global namespace:

ConformalLayout.ep	
ConformalLayout.en	
ConformalLayout.eo	
ConformalLayout.einf	
ConformalLayout.E0	
ConformalLayout.I_base	
up(x)	up-project a vector from GA to CGA
down(x)	down-project a vector from CGA to GA
homo(x)	homogenize a CGA vector

Examples

```
>>> from clifford import Cl, conformalize
>>> G2, blades = Cl(2)
>>> G2c, bladesc, stuff = conformalize(G2)
>>> locals().update(bladesc)
>>> locals().update(stuff)
```

Whether you construct your algebras from scratch, or use the predefined ones, you'll end up working with the following types:

MultiVector(layout[, value, string, dtype])	An element of the algebra
Layout(sig, *[, ids, order, names])	Layout stores information regarding the geometric algebra itself and the internal representation of multivectors.
ConformalLayout(*args[, layout])	A layout for a conformal algebra, which adds extra constants and helpers.

clifford.MultiVector

class clifford.**MultiVector**(*layout*, *value=None*, *string=None*, ***, *dtype: numpy.dtype = <class 'numpy.float64'>*)

An element of the algebra

Parameters

- **layout** (instance of *clifford.Layout*) – The layout of the algebra
- **value** (sequence of length `layout.gaDims`) – The coefficients of the base blades
- **dtype** (*numpy.dtype*) – The datatype to use for the multivector, if no value was passed.

  New in version 1.1.0.

Notes

The following operators are overloaded:

- `A * B` : geometric product
- `A ^ B` : outer product
- `A | B` : inner product
- `A << B` : left contraction
- `~M` : reversion
- `M(N)` : grade or subspace projection
- `M[N]` : blade projection

exp() → clifford._multivector.MultiVector

vee(*other*) → clifford._multivector.MultiVector
Vee product $A \vee B$.

This is often defined as:

$$(A \vee B)^* = A^* \wedge B^*$$
$$\implies A \vee B = (A^* \wedge B^*)^{-*}$$

This is very similar to the *meet()* function, but always uses the dual in the full space .

Internally, this is actually implemented using the complement functions instead, as these work in degenerate metrics like PGA too, and are equivalent but faster in other metrics.

__and__(*other*) → clifford._multivector.MultiVector
`self & other`, an alias for *vee()*

__mul__(*other*) → clifford._multivector.MultiVector
`self * other`, the geometric product MN

__xor__(*other*) → clifford._multivector.MultiVector
`self ^ other`, the Outer product $M \wedge N$

__or__(*other*) → clifford._multivector.MultiVector
`self | other`, the inner product $M \cdot N$

__add__(*other*) → clifford._multivector.MultiVector
`self + other`, addition

__sub__(*other*) → clifford._multivector.MultiVector
`self - other`, Subtraction

right_complement() → clifford._multivector.MultiVector

left_complement() → clifford._multivector.MultiVector

as_array() → numpy.ndarray

mag2() → numbers.Number
Magnitude (modulus) squared, $|M|^2$

Note in mixed signature spaces this may be negative

adjoint() → clifford._multivector.MultiVector
Adjoint / reversion, $\tilde{M}$

Aliased as `~M` to reflect $\tilde{M}$, one of several conflicting notations.

Note that ~(N * M) == ~M * ~N.

__invert__() → clifford._multivector.MultiVector
Adjoint / reversion, $\tilde{M}$

Aliased as ~M to reflect $\tilde{M}$, one of several conflicting notations.

Note that ~(N * M) == ~M * ~N.

__getitem__(*key: Union[MultiVector, tuple, int]*) → numbers.Number
value = self[key].

If key is a blade tuple (e.g. (0, 1) or (1, 3)), or a blade, (e.g. e12), then return the (real) value of that blade's coefficient. Otherwise, treat key as an index into the list of coefficients.

__call__(*other*, **others*) → clifford._multivector.MultiVector
Return a new multi-vector projected onto a grade or another MultiVector

M(g1, ... gn) gives $\langle M \rangle_{g1} + \cdots + \langle M \rangle_{gn}$

M(N) calls *project()* as N.project(M).

Examples

```
>>> from clifford.g2 import *
>>> M = 1 + 2*e1 + 3*e12
>>> M(0)
1
>>> M(0, 2)
1 + (3^e12)
```

clean(*eps=None*) → clifford._multivector.MultiVector
Sets coefficients whose absolute value is < eps to exactly 0.

eps defaults to the current value of the global _settings._eps.

round(*eps=None*) → clifford._multivector.MultiVector
Rounds all coefficients according to Python's rounding rules.

eps defaults to the current value of the global _settings._eps.

lc(*other*) → clifford._multivector.MultiVector
The left-contraction of two multivectors, $M \rfloor N$

property pseudoScalar
Returns a MultiVector that is the pseudoscalar of this space.

property I
Returns a MultiVector that is the pseudoscalar of this space.

invPS() → clifford._multivector.MultiVector
Returns the inverse of the pseudoscalar of the algebra.

isScalar() → bool
Returns true iff self is a scalar.

isBlade() → bool
Returns true if multivector is a blade.

isVersor() → bool
Returns true if multivector is a versor. From Leo Dorsts GA for computer science section 21.5, definition from 7.6.4

grades(*eps=None*) → Set[int]
Return the grades contained in the multivector.

Changed in version 1.1.0: Now returns a set instead of a list

Changed in version 1.3.0: Accepts an *eps* argument

property blades_list
ordered list of blades present in this MV

normal() → clifford._multivector.MultiVector
Return the (mostly) normalized multivector.

The _mostly_ comes from the fact that some multivectors have a negative squared-magnitude. So, without introducing formally imaginary numbers, we can only fix the normalized multivector's magnitude to +-1.

$\frac{M}{|M|}$ up to a sign

leftLaInv() → clifford._multivector.MultiVector
Return left-inverse using a computational linear algebra method proposed by Christian Perwass.

normalInv(*check=True*) → clifford._multivector.MultiVector
The inverse of itself if $M\tilde{M} = |M|^2$.

$$M^{-1} = \tilde{M}/(M\tilde{M})$$

Parameters **check** (*bool*) – When true, the default, validate that it is appropriate to use this method of inversion.

inv() → clifford._multivector.MultiVector

leftInv() → clifford._multivector.MultiVector
Return left-inverse using a computational linear algebra method proposed by Christian Perwass.

rightInv() → clifford._multivector.MultiVector
Return left-inverse using a computational linear algebra method proposed by Christian Perwass.

dual(*I=None*) → clifford._multivector.MultiVector
The dual of the multivector against the given subspace I, $\tilde{M} = MI^{-1}$

I defaults to the pseudoscalar.

commutator(*other*) → clifford._multivector.MultiVector
The commutator product of two multivectors.

$[M, N] = M \times N = (MN + NM)/2$

x(*other*) → clifford._multivector.MultiVector
The commutator product of two multivectors.

$[M, N] = M \times N = (MN + NM)/2$

anticommutator(*other*) → clifford._multivector.MultiVector
The anti-commutator product of two multivectors, $(MN + NM)/2$

gradeInvol() → clifford._multivector.MultiVector
The grade involution of the multivector.

$$M^* = \sum_{i=0}^{\text{dims}} (-1)^i \langle M \rangle_i$$

property even
Even part of this multivector

defined as `M + M.gradInvol()`

property odd
Odd part of this mulitvector

defined as `M +- M.gradInvol()`

conjugate() → clifford._multivector.MultiVector
The Clifford conjugate (reversion and grade involution).

M^* = `(~M).gradeInvol()`

project(*other*) → clifford._multivector.MultiVector
Projects the multivector onto the subspace represented by this blade.

$P_A(M) = (M \rfloor A)A^{-1}$

factorise() → Tuple[List[clifford._multivector.MultiVector], numbers.Number]
Factorises a blade into basis vectors and an overall scale.

Uses Leo Dorsts algorithm from 21.6 of GA for Computer Science

basis() → List[clifford._multivector.MultiVector]
Finds a vector basis of this subspace.

join(*other*) → clifford._multivector.MultiVector
The join of two blades, $J = A \cup B$

Similar to the wedge, $W = A \wedge B$, but without decaying to 0 for blades which share a vector.

meet(*other*, *subspace=None*) → clifford._multivector.MultiVector
The meet of two blades, $A \cap B$.

Computation is done with respect to a subspace that defaults to the *join()* if none is given.

Similar to the *vee()*, $V = A \vee B$, but without decaying to 0 for blades lying in the same subspace.

astype(**args*, ***kwargs*)
Change the underlying scalar type of this vector.

Can be used to force lower-precision floats or integers

See *np.ndarray.astype* for argument descriptions.

clifford.Layout

class clifford.**Layout**(*sig*, *, *ids=None*, *order=None*, *names=None*)
Layout stores information regarding the geometric algebra itself and the internal representation of multivectors.

Parameters

- **sig** (*List[int]*) – The signature of the vector space. This should be a list of positive and negative numbers where the sign determines the sign of the inner product of the corresponding vector with itself. The values are irrelevant except for sign. This list also determines the dimensionality of the vectors.

  Examples:

```
sig=[+1, -1, -1, -1] # Hestenes', et al. Space-Time Algebra
sig=[+1, +1, +1]     # 3-D Euclidean signature
```

- **ids** (*Optional[BasisVectorIds[Any]]*) – A list of ids to associate with each basis vector. These ids are used to generate names (if not passed explicitly), and also used when using tuple-notation to access elements, such as `mv[(1, 3)] = 1`. Defaults to `BasisVectorIds.ordered_integers(len(sig))`; that is, integers starting at 1. This supersedes the old *firstIdx* argument.

  Examples:

  ```
  ids=BasisVectorIds.ordered_integers(2, first_index=1)
  ids=BasisVectorIds([10, 20, 30])
  ```

  New in version 1.3.0.

- **order** (*Optional[BasisBladeOrder]*) – A specification of the memory order to use when storing the basis blades. Defaults to `BasisBladeOrder.shortlex(len(sig))`. This supersedes the old *bladeTupList* argument.

  Warning: Various tools within clifford assume this default, so do not change this unless you know what you're doing!

  New in version 1.3.0.

- **bladeTupList** (*List[Tuple[int, ..]]*) – List of tuples corresponding to the blades in the whole algebra. This list determines the order of coefficients in the internal representation of multivectors. The entry for the scalar must be an empty tuple, and the entries for grade-1 vectors must be singleton tuples. Remember, the length of the list will be ``2**dims`.

  Example:

  ```
  bladeTupList = [(), (0,), (1,), (0, 1)]  # 2-D
  ```

  Deprecated since version 1.3.0: Use the new *order* and *ids* arguments instead. The above example can be spelt with the slightly longer:

  ```
  ids = BasisVectorIds([.ordered_integers(2, first_index=0)])
  order = ids.order_from_tuples([(), (0,), (1,), (0, 1)])
  Layout(sig, ids=ids, order=order)
  ```

- **firstIdx** (*int*) – The index of the first vector. That is, some systems number the base vectors starting with 0, some with 1.

  Deprecated since version 1.3.0: Use the new *ids* argument instead, for which the docs show an equivalent replacement

- **names** (*List[str]*) – List of names of each blade. When pretty-printing multivectors, use these symbols for the blades. names should be in the same order as *order*. You may use an empty string for scalars. By default, the name for each non-scalar blade is 'e' plus the ids of the blade as given in *ids*.

  Example:

  ```
  names=['', 's0', 's1', 'i']  # 2-D
  ```

dims
: dimensionality of vectors (`len(self.sig)`)

sig
normalized signature, with all values `+1` or `-1`

bladeTupList
list of blades

gradeList
corresponding list of the grades of each blade

gaDims
2**dims

names
pretty-printing symbols for the blades

gmt
Multiplication table for the geometric product.

This is a tensor of rank 3 such that $a = bc$ can be computed as $a_j = \sum_{i,k} b_i M_{ijk} c_k$.

omt
Multiplication table for the inner product, stored in the same way as *gmt*

imt
Multiplication table for the outer product, stored in the same way as *gmt*

lcmt
Multiplication table for the left-contraction, stored in the same way as *gmt*

bladeTupList

property firstIdx
Starting point for vector indices

Deprecated since version 1.3.0: This attribute has been deprecated, to match the deprecation of the matching argument in the constructor. Internal code should be using `self._basis_vector_ids.values[x]` instead of `x + self.firstIdx`. This replacement API is not yet finalized, so if you need it please file an issue on github!

dual_func
Generates the dual function for the pseudoscalar

vee_func
Generates the vee product function

parse_multivector(*mv_string: str*) → clifford._multivector.MultiVector
Parses a multivector string into a MultiVector object

gmt_func_generator(*grades_a=None*, *grades_b=None*, *filter_mask=None*)

imt_func_generator(*grades_a=None*, *grades_b=None*, *filter_mask=None*)

omt_func_generator(*grades_a=None*, *grades_b=None*, *filter_mask=None*)

lcmt_func_generator(*grades_a=None*, *grades_b=None*, *filter_mask=None*)

get_grade_projection_matrix(*grade*)
Returns the matrix M_g that performs grade projection via left multiplication eg. `M_g@A.value = A(g).value`

gmt_func

imt_func

omt_func

lcmt_func

left_complement_func

right_complement_func

adjoint_func
This function returns a fast jitted adjoint function

inv_func
Get a function that returns left-inverse using a computational linear algebra method proposed by Christian Perwass.

-1 -1

M where M * M == 1

get_left_gmt_matrix(*x*)
This produces the matrix X that performs left multiplication with x eg. `X@b == (x*b).value`

get_right_gmt_matrix(*x*)
This produces the matrix X that performs right multiplication with x eg. `X@b == (b*x).value`

load_ga_file(*filename: str*) → clifford._mvarray.MVArray
Loads the data from a ga file, checking it matches this layout.

grade_mask(*grade: int*) → numpy.ndarray

property rotor_mask

property metric

property scalar
the scalar of value 1, for this GA (a MultiVector object)

useful for forcing a MultiVector type

property pseudoScalar
the psuedoScalar

property I
the psuedoScalar

randomMV(*n=1*, ***kwargs*) → clifford._multivector.MultiVector
Convenience method to create a random multivector.

see *clifford.randomMV* for details

randomV(*n=1*, ***kwargs*) → clifford._multivector.MultiVector
generate n random 1-vector s

randomRotor() → clifford._multivector.MultiVector
generate a random Rotor.

this is created by muliplying an N unit vectors, where N is the dimension of the algebra if its even; else its one less.

property basis_vectors
dictionary of basis vectors

property basis_names
Get the names of the basis vectors, in the order they are stored.

Changed in version 1.3.0: Returns a list instead of a numpy array

property basis_vectors_lst
Like `blades_of_grade(1)`, but ordered based on the `ids` parameter passed at construction.

blades_of_grade(*grade: int*) → List[clifford._multivector.MultiVector]
return all blades of a given grade,

property blades_list
List of blades in this layout matching the *order* argument this layout was constructed from.

property blades

bases(*mvClass=<class 'clifford._multivector.MultiVector'>*, *grades: Optional[Container[int]] = None*) → Dict[str, clifford._multivector.MultiVector]
Returns a dictionary mapping basis element names to their MultiVector instances, optionally for specific grades

if you are lazy, you might do this to populate your namespace with the variables of a given layout.

```
>>> locals().update(layout.blades())
```

Changed in version 1.1.0: This dictionary includes the scalar

MultiVector(**args*, ***kwargs*) → clifford._multivector.MultiVector
Create a multivector in this layout

convenience func to `MultiVector(layout)`

clifford.ConformalLayout

class clifford.**ConformalLayout**(**args*, *layout=None*, ***kwargs*)
Bases: `clifford._layout.Layout`

A layout for a conformal algebra, which adds extra constants and helpers.

Typically these should be constructed via *clifford.conformalize()*.

New in version 1.2.0.

ep
The first added basis element, e_+, usually with $e_+^2 = +1$

Type *MultiVector*

en
The first added basis element, e_-, usually with $e_-^2 = -1$

Type *MultiVector*

eo
The null basis vector at the origin, $e_o = 0.5(e_- - e_+)$

Type *MultiVector*

einf
The null vector at infinity, $e_\infty = e_- + e_+$

Type *MultiVector*

E0
The minkowski subspace bivector, $e_\infty \wedge e_o$

Type *MultiVector*

I_base
The pseudoscalar of the base ga, in cga layout

Type *MultiVector*

up (*x: clifford._multivector.MultiVector*) → clifford._multivector.MultiVector
up-project a vector from GA to CGA

homo (*x: clifford._multivector.MultiVector*) → clifford._multivector.MultiVector
homogenize a CGA vector

down (*x: clifford._multivector.MultiVector*) → clifford._multivector.MultiVector
down-project a vector from CGA to GA

Advanced algebra configuration

It is unlikely you will need these features, but they remain as a better spelling for features which have always been in `clifford`.

`BasisBladeOrder`(bitmaps)	Represents the storage order in memory of basis blade coefficients.
`BasisVectorIds`(blade_ids)	Stores ids for the ordered set of basis vectors, typically integers.

1.1.2 Global configuration functions

These functions are used to change the global behavior of `clifford`.

`clifford.`**eps** (*newEps=None*)
Get/Set the epsilon for float comparisons.

`clifford.`**pretty** (*precision=None*)
Makes `repr(MultiVector)` default to pretty-print.

precision arg can be used to set the printed precision.

Parameters **precision** (*int*) – number of sig figs to print past decimal

Examples

```
>>> pretty(5)
```

`clifford.`**ugly**()
Makes `repr(MultiVector)` default to eval-able representation.

`clifford.`**print_precision** (*newVal*)
Set the epsilon for float comparisons.

Parameters **newVal** (*int*) – number of sig figs to print (see builtin *round*)

Examples

```
>>> print_precision(5)
```

1.1.3 Miscellaneous classes

MVArray	MultiVector Array
Frame	A frame of vectors
BladeMap(blades_map[, map_scalars])	A Map Relating Blades in two different algebras

clifford.Frame

class clifford.**Frame**
Bases: clifford._mvarray.MVArray

A frame of vectors

property En
Volume element for this frame, $E_n = e_1 \wedge e_2 \wedge \cdots \wedge e_n$

property inv
The inverse frame of self

is_innermorphic_to(*other: clifford._frame.Frame*, *eps: float = None*) → bool
Is this frame *innermorphic* to other?

innermorphic means both frames share the same inner-product between corresponding vectors. This implies that the two frames are related by an orthogonal transform.

clifford.BladeMap

class clifford.**BladeMap**(*blades_map*, *map_scalars=True*)
A Map Relating Blades in two different algebras

Examples

```
>>> from clifford import Cl
>>> # Dirac Algebra  `D`
>>> D, D_blades = Cl(1, 3, firstIdx=0, names='d')
>>> locals().update(D_blades)
```

```
>>> # Pauli Algebra  `P`
>>> P, P_blades = Cl(3, names='p')
>>> locals().update(P_blades)
>>> sta_split = BladeMap([(d01, p1),
...                       (d02, p2),
...                       (d03, p3),
...                       (d12, p12),
...                       (d23, p23),
...                       (d13, p13)])
```

property b1

property b2

property layout1

property layout2

1.1.4 Miscellaneous functions

grade_obj(objin[, threshold])	Returns the modal grade of a multivector
randomMV(layout[, min, max, grades, ...])	n Random MultiVectors with given layout.

clifford.grade_obj

clifford.**grade_obj**(*objin*, *threshold=1e-07*)

Returns the modal grade of a multivector

clifford.randomMV

clifford.**randomMV**(*layout*, *min=-2.0*, *max=2.0*, *grades=None*, *mvClass=<class 'clifford._multivector.MultiVector'>*, *uniform=None*, *n=1*, *normed=False*)

n Random MultiVectors with given layout.

Coefficients are between min and max, and if grades is a list of integers, only those grades will be non-zero.

Examples

```
>>> randomMV(layout, min=-2.0, max=2.0, grades=None, uniform=None, n=2)
```

1.2 cga (clifford.cga)

Object Oriented Conformal Geometric Algebra.

Examples

```
>>> from clifford import Cl
>>> from clifford.cga import CGA
>>> g3, blades = Cl(3)
>>> locals().update(blades)
>>> g3c = CGA(g3)
>>> C = g3c.round(3)              # create random sphere
>>> T = g3c.translation(e1+e2)    # create translation
>>> C_ = T(C)                     # translate the sphere
>>> C_.center                     # compute center of sphere
-(1.0^e4) - (1.0^e5)
```

1.2.1 The CGA

CGA(layout_orig)	Conformal Geometric Algebra

clifford.cga.CGA

class clifford.cga.**CGA**(*layout_orig*)
Conformal Geometric Algebra

conformalizes the layout_orig, and provides several methods and for objects/operators

Parameters **layout_orig** ([*clifford.Layout*, int]) – a layout for the *base* geometric algebra which is conformalized if given as an int, then generates a euclidean space of given dimension

Examples

```
>>> from clifford import Cl
>>> from clifford.cga import CGA
>>> g3, blades  = Cl(3)
>>> g3c = CGA(g3)
>>> g3c = CGA(3)
```

Methods

__init__	Initialize self.
base_vector	random vector in the lower(original) space
dilation	see *Dilation*
flat	see *Flat*
null_vector	generates random null vector if x is None, or returns a null vector from base vector x, if x^self.I_base ==0 returns x,
rotation	see *Rotation*
round	see *Round*
straight_up	place a vector from layout_orig into this CGA, without up()
translation	see *Translation*
transversion	see *Transversion*

clifford.cga.CGA.__init__

CGA.**__init__**(*layout_orig*) → None
Initialize self. See help(type(self)) for accurate signature.

clifford.cga.CGA.base_vector

CGA.**base_vector**() → clifford._multivector.MultiVector
random vector in the lower(original) space

clifford.cga.CGA.dilation

CGA.**dilation**(**args*) → *clifford.cga.Dilation*
see *Dilation*

clifford.cga.CGA.flat

CGA.**flat**(**args*) → *clifford.cga.Flat*
see *Flat*

clifford.cga.CGA.null_vector

CGA.**null_vector**(*x=None*) → clifford._multivector.MultiVector
generates random null vector if x is None, or returns a null vector from base vector x, if x^self.I_base ==0 returns x,

a null vector will lay on the horisphere

clifford.cga.CGA.rotation

CGA.**rotation**(**args*) → *clifford.cga.Rotation*
see *Rotation*

clifford.cga.CGA.round

CGA.**round**(**args*) → *clifford.cga.Round*
see *Round*

clifford.cga.CGA.straight_up

CGA.**straight_up**(*x*) → clifford._multivector.MultiVector
place a vector from layout_orig into this CGA, without up()

clifford.cga.CGA.translation

CGA.**translation**(**args*) → *clifford.cga.Translation*
see *Translation*

clifford.cga.CGA.transversion

CGA.**transversion**(**args*) → *clifford.cga.Transversion*
see *Transversion*

1.2.2 Objects

Flat(cga, *args)	A line, plane, or hyperplane.
Round(cga, *args)	A point pair, circle, sphere or hyper-sphere.

clifford.cga.Flat

class clifford.cga.**Flat**(*cga*, **args*)
A line, plane, or hyperplane.

Typically constructed as method of existing cga, like *cga.flat()*

multivector is accessable by *mv* property

Parameters

- **cga** (*CGA*) – the cga object
- **args** (*[int, Multivector, Multivectors]*) –
  - if nothing supplied, generate a flat of highest dimension
  - int: dimension of flat (2=line, 3=plane, etc)
  - Multivector : can be * existing Multivector representing the Flat * vectors on the flat

Examples

```
>>> cga = CGA(3)
>>> locals().update(cga.blades)
>>> F = cga.flat()                # from None
>>> F = cga.flat(2)               # from dim of space
>>> F = cga.flat(e1, e2)          # from points
>>> F = cga.flat(cga.flat().mv)   # from existing multivector
```

Methods

__init__	Initialize self.
inverted	inverted version of this thing.
involuted	inverted version of this thing.

clifford.cga.Flat.__init__

Flat.**__init__**(*cga*, **args*) → None
Initialize self. See help(type(self)) for accurate signature.

clifford.cga.Flat.inverted

Flat.**inverted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> ep*self*ep

where ep is the positive added basis vector

clifford.cga.Flat.involuted

Flat.**involuted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> E0*self*E0

where E0 is the added minkowski bivector

clifford.cga.Round

class clifford.cga.**Round**(*cga*, **args*)
A point pair, circle, sphere or hyper-sphere.

Typically constructed as method of existing cga, like *cga.round()*

multivector is accessable by *mv* property

Parameters

- **cga** (*CGA*) – the cga object
- **args** (*[int, Multivector, Multivectors]*) –
  - if nothing supplied, generate a round of highest dimension
  - int: dimension of flat (2=point pair, 3=circle, etc)
  - Multivector : can be * existing Multivector representing the round * vectors on the round

Examples

```
>>> cga = CGA(3)
>>> locals().update(cga.blades)
>>> cga.round()                # from None
Sphere
>>> cga.round(2)               # from dim of space
Sphere
>>> cga.round(e1, e2, -e1)     # from points
Circle
>>> cga.round(cga.flat().mv)   # from existing multivector
Sphere
```

Attributes

center	center of this round, as a null vector
center_down	center of this round, as a down-projected vector (in I_base)
dim	dimension of this round
dual	self.mv* self.layout.I
radius	radius of the round (a float)

clifford.cga.Round.center

property Round.**center**
center of this round, as a null vector

clifford.cga.Round.center_down

property Round.**center_down**
center of this round, as a down-projected vector (in I_base)

(but still in cga's layout)

clifford.cga.Round.dim

property Round.**dim**
dimension of this round

clifford.cga.Round.dual

property Round.**dual**
self.mv* self.layout.I

clifford.cga.Round.radius

property Round.**radius**
radius of the round (a float)

Methods

__init__	Initialize self.
from_center_radius	construct a round from center/radius
inverted	inverted version of this thing.
involuted	inverted version of this thing.

clifford.cga.Round.__init__

Round.**__init__**(*cga*, **args*) → None
Initialize self. See help(type(self)) for accurate signature.

clifford.cga.Round.from_center_radius

Round.**from_center_radius**(*center*, *radius*)
construct a round from center/radius

clifford.cga.Round.inverted

Round.**inverted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> ep*self*ep

where ep is the positive added basis vector

clifford.cga.Round.involuted

Round.**involuted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> E0*self*E0

where E0 is the added minkowski bivector

1.2.3 Operators

Rotation(cga, *args)	A Rotation
Dilation(cga, *args)	A global dilation
Translation(cga, *args)	A Translation
Transversion(cga, *args)	A Transversion

clifford.cga.Rotation

class clifford.cga.**Rotation**(*cga*, **args*)

A Rotation

Can be constructed from a generator, rotor, or none

Parameters **args** ([none, *clifford.Multivector*]) – if none, a random translation will be generated several types of Multivectors can be used:

- bivector - interpreted as the generator
- existing translation rotor

Examples

```
>>> cga = CGA(3)
>>> locals().update(cga.blades)
>>> R = cga.rotation()           # from None
>>> R = cga.rotation(e12+e23)    # from bivector
>>> R = cga.rotation(R.mv)    # from bivector
```

Methods

__init__	Initialize self.
inverted	inverted version of this thing.
involuted	inverted version of this thing.

clifford.cga.Rotation.__init__

Rotation.**__init__**(*cga*, **args*) → None

Initialize self. See help(type(self)) for accurate signature.

clifford.cga.Rotation.inverted

Rotation.**inverted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> ep*self*ep

where ep is the positive added basis vector

clifford.cga.Rotation.involuted

Rotation.**involuted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> E0*self*E0

where E0 is the added minkowski bivector

clifford.cga.Dilation

class clifford.cga.**Dilation**(*cga*, **args*)
A global dilation

Parameters **args** ([*none*, *number*]) – if none, a random dilation will be generated if a number, dilation of given amount

Examples

```
>>> cga = CGA(3)
>>> D = cga.dilation()          # from  none
>>> D = cga.dilation(.4)        # from number
```

Methods

__init__	Initialize self.
inverted	inverted version of this thing.
involuted	inverted version of this thing.

clifford.cga.Dilation.__init__

Dilation.**__init__**(*cga*, **args*) → None
Initialize self. See help(type(self)) for accurate signature.

clifford.cga.Dilation.inverted

Dilation.**inverted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> ep*self*ep

where ep is the positive added basis vector

clifford.cga.Dilation.involuted

Dilation.**involuted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> E0*self*E0

where E0 is the added minkowski bivector

clifford.cga.Translation

class clifford.cga.**Translation**(*cga*, **args*)
A Translation

Can be constructed from a vector in base space or a null vector, or nothing.

Parameters **args** ([none, *clifford.Multivector*]) – if none, a random translation will be generated several types of Multivectors can be used:

- base vector - vector in base space
- null vector
- existing translation rotor

Examples

```
>>> cga = CGA(3)
>>> locals().update(cga.blades)
>>> T = cga.translation()        # from None
>>> T = cga.translation(e1+e2)   # from base vector
>>> T = cga.translation(cga.up(e1+e2)) # from null vector
>>> T = cga.translation(T.mv)   # from existing translation rotor
```

Methods

__init__	Initialize self.
inverted	inverted version of this thing.
involuted	inverted version of this thing.

clifford.cga.Translation.__init__

Translation.**__init__**(*cga*, **args*) → None
Initialize self. See help(type(self)) for accurate signature.

clifford.cga.Translation.inverted

Translation.**inverted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> ep*self*ep

where ep is the positive added basis vector

clifford.cga.Translation.involuted

Translation.**involuted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> E0*self*E0

where E0 is the added minkowski bivector

clifford.cga.Transversion

class clifford.cga.**Transversion**(*cga*, **args*)
A Transversion

A transversion is a combination of an inversion-translation-inversion, or in other words an inverted translation operator. This inherits from *Translation*

Can be constructed from a vector in base space or a null vector, or nothing.

Parameters **args** ([none, *clifford.Multivector*]) – if none, a random transversion will be generated several types of Multivectors can be used:

- base vector - vector in base space
- null vector
- existing transversion rotor

Examples

```
>>> cga = CGA(3)
>>> locals().update(cga.blades)
>>> K = cga.transversion()        # from None
>>> K = cga.transversion(e1+e2)   # from base vector
>>> K = cga.transversion(cga.up(e1+e2)) # from null vector
>>> T = cga.translation()
>>> K = cga.transversion(T.mv)   # from existing translation rotor
```

Methods

__init__	Initialize self.
inverted	inverted version of this thing.
involuted	inverted version of this thing.

clifford.cga.Transversion.__init__

Transversion.**__init__**(*cga*, **args*) → None
Initialize self. See help(type(self)) for accurate signature.

clifford.cga.Transversion.inverted

Transversion.**inverted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> ep*self*ep

where ep is the positive added basis vector

clifford.cga.Transversion.involuted

Transversion.**involuted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> E0*self*E0

where E0 is the added minkowski bivector

1.2.4 Meta-Class

CGAThing(cga)	base class for cga objects and operators.

clifford.cga.CGAThing

class clifford.cga.**CGAThing**(*cga:* clifford.cga.CGA)
base class for cga objects and operators.

maps versor product to *__call__*.

Methods

__init__	Initialize self.
inverted	inverted version of this thing.
involuted	inverted version of this thing.

clifford.cga.CGAThing.__init__

CGAThing.**__init__**(*cga:* clifford.cga.CGA) → None
Initialize self. See help(type(self)) for accurate signature.

clifford.cga.CGAThing.inverted

CGAThing.**inverted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> ep*self*ep

where ep is the positive added basis vector

clifford.cga.CGAThing.involuted

CGAThing.**involuted**() → clifford._multivector.MultiVector
inverted version of this thing.

self -> E0*self*E0

where E0 is the added minkowski bivector

1.3 tools (clifford.tools)

Algorithms and tools of various kinds.

1.3.1 Tools for specific ga's

g3	Tools for 3DGA (g3)
g3c	Tools for 3DCGA (g3c)

clifford.tools.g3

Tools for 3DGA (g3)

3DGA Tools

Rotation Conversion Methods

quaternion_to_rotor(quaternion)	Converts a quaternion into a pure rotation rotor
rotor_to_quaternion(R)	Converts a pure rotation rotor into a quaternion
quaternion_to_matrix(q)	Converts a quaternion into a rotation matrix
rotation_matrix_to_quaternion(a)	Converts a rotation matrix into a quaternion

clifford.tools.g3.quaternion_to_rotor

clifford.tools.g3.**quaternion_to_rotor**(*quaternion*)
Converts a quaternion into a pure rotation rotor

clifford.tools.g3.rotor_to_quaternion

clifford.tools.g3.**rotor_to_quaternion**(*R*)
Converts a pure rotation rotor into a quaternion

clifford.tools.g3.quaternion_to_matrix

clifford.tools.g3.**quaternion_to_matrix**(*q*)
Converts a quaternion into a rotation matrix

clifford.tools.g3.rotation_matrix_to_quaternion

clifford.tools.g3.**rotation_matrix_to_quaternion**(*a*)
Converts a rotation matrix into a quaternion

Generation Methods

random_unit_vector()	Creates a random unit vector
random_euc_mv([l_max])	Creates a random vector normally distributed with length l_max
generate_rotation_rotor(theta, euc_vector_m, ...)	Generates a rotation of angle theta in the m, n plane
random_rotation_rotor([max_angle])	Creates a random rotation rotor

clifford.tools.g3.random_unit_vector

clifford.tools.g3.**random_unit_vector**()
Creates a random unit vector

clifford.tools.g3.random_euc_mv

clifford.tools.g3.**random_euc_mv**(*l_max=10*)
Creates a random vector normally distributed with length l_max

clifford.tools.g3.generate_rotation_rotor

clifford.tools.g3.**generate_rotation_rotor**(*theta*, *euc_vector_m*, *euc_vector_n*)
Generates a rotation of angle theta in the m, n plane

clifford.tools.g3.random_rotation_rotor

clifford.tools.g3.**random_rotation_rotor**(*max_angle=3.141592653589793*)
Creates a random rotation rotor

Misc

angle_between_vectors(v1, v2)	Returns the angle between two conformal vectors
np_to_euc_mv(np_in)	Converts a 3d numpy vector to a 3d GA point
euc_mv_to_np(euc_point)	Converts a 3d GA point to a 3d numpy vector
euc_cross_prod(euc_a, euc_b)	Implements the cross product in GA
rotor_vector_to_vector(v1, v2)	Creates a rotor that takes one vector into another
correlation_matrix(u_list, v_list)	Creates a correlation matrix between vector lists
GA_SVD(u_list, v_list)	Does SVD on a pair of GA vectors
rotation_matrix_align_vecs(u_list, v_list)	Returns the rotation matrix that aligns the set of vectors u and v
rotor_align_vecs(u_list, v_list)	Returns the rotation rotor that aligns the set of vectors u and v

clifford.tools.g3.angle_between_vectors

clifford.tools.g3.**angle_between_vectors**(*v1*, *v2*)
Returns the angle between two conformal vectors

clifford.tools.g3.np_to_euc_mv

clifford.tools.g3.**np_to_euc_mv**(*np_in*)
Converts a 3d numpy vector to a 3d GA point

clifford.tools.g3.euc_mv_to_np

clifford.tools.g3.**euc_mv_to_np**(*euc_point*)
Converts a 3d GA point to a 3d numpy vector

clifford.tools.g3.euc_cross_prod

clifford.tools.g3.**euc_cross_prod**(*euc_a*, *euc_b*)
Implements the cross product in GA

clifford.tools.g3.rotor_vector_to_vector

clifford.tools.g3.**rotor_vector_to_vector**(*v1*, *v2*)
Creates a rotor that takes one vector into another

clifford.tools.g3.correlation_matrix

clifford.tools.g3.**correlation_matrix**(*u_list*, *v_list*)
Creates a correlation matrix between vector lists

clifford.tools.g3.GA_SVD

clifford.tools.g3.**GA_SVD**(*u_list*, *v_list*)
Does SVD on a pair of GA vectors

clifford.tools.g3.rotation_matrix_align_vecs

clifford.tools.g3.**rotation_matrix_align_vecs**(*u_list*, *v_list*)
Returns the rotation matrix that aligns the set of vectors u and v

clifford.tools.g3.rotor_align_vecs

clifford.tools.g3.**rotor_align_vecs**(*u_list*, *v_list*)
Returns the rotation rotor that aligns the set of vectors u and v

clifford.tools.g3c

Tools for 3DCGA (g3c)

3DCGA Tools

Generation Methods

random_bivector()	Creates a random bivector on the form described by R.
standard_point_pair_at_origin()	Creates a standard point pair at the origin
random_point_pair_at_origin()	Creates a random point pair bivector object at the origin
random_point_pair()	Creates a random point pair bivector object
standard_line_at_origin()	Creates a standard line at the origin
random_line_at_origin()	Creates a random line at the origin
random_line()	Creates a random line
random_circle_at_origin()	Creates a random circle at the origin
random_circle()	Creates a random circle
random_sphere_at_origin()	Creates a random sphere at the origin
random_sphere()	Creates a random sphere
random_plane_at_origin()	Creates a random plane at the origin
random_plane()	Creates a random plane
generate_n_clusters(object_generator, ...)	Creates n_clusters of random objects
generate_random_object_cluster(n_objects, ...)	Creates a cluster of random objects
random_translation_rotor([maximum_translation])	generate a random translation rotor
random_rotation_translation_rotor([...])	generate a random combined rotation and translation rotor
random_conformal_point([l_max])	Creates a random conformal point
generate_dilation_rotor(scale)	Generates a rotor that performs dilation about the origin
generate_translation_rotor(euc_vector_a)	Generates a rotor that translates objects along the euclidean vector euc_vector_a

clifford.tools.g3c.random_bivector

clifford.tools.g3c.**random_bivector**()

Creates a random bivector on the form described by R. Wareham in Mesh Vertex Pose and Position Interpolation using Geometric Algebra. $$ B = ab + c*n_{inf}$$ where $a, b, c in mathcal(R)^3$

clifford.tools.g3c.standard_point_pair_at_origin

clifford.tools.g3c.**standard_point_pair_at_origin**()
Creates a standard point pair at the origin

clifford.tools.g3c.random_point_pair_at_origin

clifford.tools.g3c.**random_point_pair_at_origin**()
Creates a random point pair bivector object at the origin

clifford.tools.g3c.random_point_pair

clifford.tools.g3c.**random_point_pair**()
Creates a random point pair bivector object

clifford.tools.g3c.standard_line_at_origin

clifford.tools.g3c.**standard_line_at_origin**()
Creates a standard line at the origin

clifford.tools.g3c.random_line_at_origin

clifford.tools.g3c.**random_line_at_origin**()
Creates a random line at the origin

clifford.tools.g3c.random_line

clifford.tools.g3c.**random_line**()
Creates a random line

clifford.tools.g3c.random_circle_at_origin

clifford.tools.g3c.**random_circle_at_origin**()
Creates a random circle at the origin

clifford.tools.g3c.random_circle

clifford.tools.g3c.**random_circle**()
Creates a random circle

clifford.tools.g3c.random_sphere_at_origin

clifford.tools.g3c.**random_sphere_at_origin**()
Creates a random sphere at the origin

clifford.tools.g3c.random_sphere

clifford.tools.g3c.**random_sphere**()
Creates a random sphere

clifford.tools.g3c.random_plane_at_origin

clifford.tools.g3c.**random_plane_at_origin**()
Creates a random plane at the origin

clifford.tools.g3c.random_plane

clifford.tools.g3c.**random_plane**()
Creates a random plane

clifford.tools.g3c.generate_n_clusters

clifford.tools.g3c.**generate_n_clusters**(*object_generator*, *n_clusters*, *n_objects_per_cluster*)
Creates n_clusters of random objects

clifford.tools.g3c.generate_random_object_cluster

clifford.tools.g3c.**generate_random_object_cluster**(*n_objects*, *object_generator*, *max_cluster_trans=1.0*, *max_cluster_rot=0.39269908169872414*)
Creates a cluster of random objects

clifford.tools.g3c.random_translation_rotor

clifford.tools.g3c.**random_translation_rotor**(*maximum_translation=10.0*)
generate a random translation rotor

clifford.tools.g3c.random_rotation_translation_rotor

clifford.tools.g3c.**random_rotation_translation_rotor**(*maximum_translation=10.0*, *maximum_angle=3.141592653589793*)

generate a random combined rotation and translation rotor

clifford.tools.g3c.random_conformal_point

clifford.tools.g3c.**random_conformal_point**(*l_max=10*)

Creates a random conformal point

clifford.tools.g3c.generate_dilation_rotor

clifford.tools.g3c.**generate_dilation_rotor**(*scale*)

Generates a rotor that performs dilation about the origin

clifford.tools.g3c.generate_translation_rotor

clifford.tools.g3c.**generate_translation_rotor**(*euc_vector_a*)

Generates a rotor that translates objects along the euclidean vector euc_vector_a

Geometry Methods

intersect_line_and_plane_to_point(line, plane)	Returns the point at the intersection of a line and plane If there is no intersection it returns None
val_intersect_line_and_plane_to_point(...)	Returns the point at the intersection of a line and plane
quaternion_and_vector_to_rotor(quaternion, ...)	Takes in a quaternion and a vector and returns a conformal rotor that implements the transformation
get_center_from_sphere(sphere)	Returns the conformal point at the centre of a sphere by reflecting the point at infinity
get_radius_from_sphere(sphere)	Returns the radius of a sphere
point_pair_to_end_points(T)	Extracts the end points of a point pair bivector
val_point_pair_to_end_points(T)	Extracts the end points of a point pair bivector
get_circle_in_euc(circle)	Extracts all the normal stuff for a circle
circle_to_sphere(C)	returns the sphere for which the input circle is the perimeter
line_to_point_and_direction(line)	converts a line to the conformal nearest point to the origin and a euc direction vector in direction of the line
get_plane_origin_distance(plane)	Get the distance between a given plane and the origin
get_plane_normal(plane)	Get the normal to the plane
get_nearest_plane_point(plane)	Get the nearest point to the origin on the plane
val_convert_2D_polar_line_to_conformal_line(...)	Converts a 2D polar line to a conformal line
convert_2D_polar_line_to_conformal_line(...)	Converts a 2D polar line to a conformal line
val_convert_2D_point_to_conformal(x, y)	Convert a 2D point to conformal
convert_2D_point_to_conformal(x, y)	Convert a 2D point to conformal

continues on next page

Table 25 – continued from previous page

val_distance_point_to_line(point, line)	Returns the euclidean distance between a point and a line
distance_polar_line_to_euc_point_2d(rho, ...)	Return the distance between a polar line and a euclidean point in 2D
midpoint_between_lines(L1, L2)	Gets the point that is maximally close to both lines Hadfield and Lasenby AGACSE2018
val_midpoint_between_lines(L1_val, L2_val)	Gets the point that is maximally close to both lines Hadfield and Lasenby AGACSE2018
midpoint_of_line_cluster(line_cluster)	Gets a center point of a line cluster Hadfield and Lasenby AGACSE2018
val_midpoint_of_line_cluster(array_line_cluster)	Gets a center point of a line cluster Hadfield and Lasenby AGACSE2018
val_midpoint_of_line_cluster_grad(...)	Gets an approximate center point of a line cluster Hadfield and Lasenby AGACSE2018
get_line_intersection(L3, Ldd)	Gets the point of intersection of two orthogonal lines that meet Xdd = Ldd*no*Ldd + no Xddd = L3*Xdd*L3 Pd = 0.5*(Xdd+Xddd) P = -(Pd*ninf*Pd)(1)/(2*(Pd|einf)**2)[0]
val_get_line_intersection(L3_val, Ldd_val)	Gets the point of intersection of two orthogonal lines that meet Xdd = Ldd*no*Ldd + no Xddd = L3*Xdd*L3 Pd = 0.5*(Xdd+Xddd) P = -(Pd*ninf*Pd)(1)/(2*(Pd|einf)**2)[0]
project_points_to_plane(point_list, plane)	Takes a load of points and projects them onto a plane
project_points_to_sphere(point_list, sphere)	Takes a load of points and projects them onto a sphere
project_points_to_circle(point_list, circle)	Takes a load of point and projects them onto a circle The closest flag determines if it should be the closest or furthest point on the circle
project_points_to_line(point_list, line)	Takes a load of points and projects them onto a line
iterative_closest_points_on_circles(C1, C2)	Given two circles C1 and C2 this calculates the closest points on each of them to the other
closest_point_on_line_from_circle(C, L[, eps])	Returns the point on the line L that is closest to the circle C Uses the algorithm described in Appendix A of Andreas Aristidou's PhD thesis
closest_point_on_circle_from_line(C, L[, eps])	Returns the point on the circle C that is closest to the line L Uses the algorithm described in Appendix A of Andreas Aristidou's PhD thesis
iterative_closest_points_circle_line(C, L[, ...])	Given a circle C and line L this calculates the closest points on each of them to the other.
iterative_furthest_points_on_circles(C1, C2)	Given two circles C1 and C2 this calculates the closest points on each of them to the other
sphere_beyond_plane(sphere, plane)	Check if the sphere is fully beyond the plane in the direction of the plane normal
sphere_behind_plane(sphere, plane)	Check if the sphere is fully behind the plane in the direction of the plane normal, ie the opposite of sphere_beyond_plane

clifford.tools.g3c.intersect_line_and_plane_to_point

clifford.tools.g3c.**intersect_line_and_plane_to_point**(*line*, *plane*)
Returns the point at the intersection of a line and plane If there is no intersection it returns None

clifford.tools.g3c.val_intersect_line_and_plane_to_point

clifford.tools.g3c.**val_intersect_line_and_plane_to_point**(*line_val*, *plane_val*)
Returns the point at the intersection of a line and plane

clifford.tools.g3c.quaternion_and_vector_to_rotor

clifford.tools.g3c.**quaternion_and_vector_to_rotor**(*quaternion*, *vector*)
Takes in a quaternion and a vector and returns a conformal rotor that implements the transformation

clifford.tools.g3c.get_center_from_sphere

clifford.tools.g3c.**get_center_from_sphere**(*sphere*)
Returns the conformal point at the centre of a sphere by reflecting the point at infinity

clifford.tools.g3c.get_radius_from_sphere

clifford.tools.g3c.**get_radius_from_sphere**(*sphere*)
Returns the radius of a sphere

clifford.tools.g3c.point_pair_to_end_points

clifford.tools.g3c.**point_pair_to_end_points**(*T*)
Extracts the end points of a point pair bivector

clifford.tools.g3c.val_point_pair_to_end_points

clifford.tools.g3c.**val_point_pair_to_end_points**(*T*)
Extracts the end points of a point pair bivector

clifford.tools.g3c.get_circle_in_euc

clifford.tools.g3c.**get_circle_in_euc**(*circle*)
Extracts all the normal stuff for a circle

clifford.tools.g3c.circle_to_sphere

clifford.tools.g3c.**circle_to_sphere**(*C*)
returns the sphere for which the input circle is the perimeter

clifford.tools.g3c.line_to_point_and_direction

clifford.tools.g3c.**line_to_point_and_direction**(*line*)
converts a line to the conformal nearest point to the origin and a euc direction vector in direction of the line

clifford.tools.g3c.get_plane_origin_distance

clifford.tools.g3c.**get_plane_origin_distance**(*plane*)
Get the distance between a given plane and the origin

clifford.tools.g3c.get_plane_normal

clifford.tools.g3c.**get_plane_normal**(*plane*)
Get the normal to the plane

clifford.tools.g3c.get_nearest_plane_point

clifford.tools.g3c.**get_nearest_plane_point**(*plane*)
Get the nearest point to the origin on the plane

clifford.tools.g3c.val_convert_2D_polar_line_to_conformal_line

clifford.tools.g3c.**val_convert_2D_polar_line_to_conformal_line**(*rho*, *theta*)
Converts a 2D polar line to a conformal line

clifford.tools.g3c.convert_2D_polar_line_to_conformal_line

clifford.tools.g3c.**convert_2D_polar_line_to_conformal_line**(*rho*, *theta*)
Converts a 2D polar line to a conformal line

clifford.tools.g3c.val_convert_2D_point_to_conformal

clifford.tools.g3c.**val_convert_2D_point_to_conformal**(*x*, *y*)
Convert a 2D point to conformal

clifford.tools.g3c.convert_2D_point_to_conformal

clifford.tools.g3c.**convert_2D_point_to_conformal**(*x*, *y*)
Convert a 2D point to conformal

clifford.tools.g3c.val_distance_point_to_line

clifford.tools.g3c.**val_distance_point_to_line**(*point*, *line*)
Returns the euclidean distance between a point and a line

clifford.tools.g3c.distance_polar_line_to_euc_point_2d

clifford.tools.g3c.**distance_polar_line_to_euc_point_2d**(*rho*, *theta*, *x*, *y*)
Return the distance between a polar line and a euclidean point in 2D

clifford.tools.g3c.midpoint_between_lines

clifford.tools.g3c.**midpoint_between_lines**(*L1*, *L2*)
Gets the point that is maximally close to both lines Hadfield and Lasenby AGACSE2018

clifford.tools.g3c.val_midpoint_between_lines

clifford.tools.g3c.**val_midpoint_between_lines**(*L1_val*, *L2_val*)
Gets the point that is maximally close to both lines Hadfield and Lasenby AGACSE2018

clifford.tools.g3c.midpoint_of_line_cluster

clifford.tools.g3c.**midpoint_of_line_cluster**(*line_cluster*)
Gets a center point of a line cluster Hadfield and Lasenby AGACSE2018

clifford.tools.g3c.val_midpoint_of_line_cluster

clifford.tools.g3c.**val_midpoint_of_line_cluster**(*array_line_cluster*)
Gets a center point of a line cluster Hadfield and Lasenby AGACSE2018

clifford.tools.g3c.val_midpoint_of_line_cluster_grad

clifford.tools.g3c.**val_midpoint_of_line_cluster_grad**(*array_line_cluster*)
Gets an approximate center point of a line cluster Hadfield and Lasenby AGACSE2018

clifford.tools.g3c.get_line_intersection

clifford.tools.g3c.**get_line_intersection**(*L3*, *Ldd*)
Gets the point of intersection of two orthogonal lines that meet Xdd = Ldd*no*Ldd + no Xddd = L3*Xdd*L3 Pd = 0.5*(Xdd+Xddd) P = -(Pd*ninf*Pd)(1)/(2*(Pd|einf)**2)[0]

clifford.tools.g3c.val_get_line_intersection

clifford.tools.g3c.**val_get_line_intersection**(*L3_val*, *Ldd_val*)
Gets the point of intersection of two orthogonal lines that meet Xdd = Ldd*no*Ldd + no Xddd = L3*Xdd*L3 Pd = 0.5*(Xdd+Xddd) P = -(Pd*ninf*Pd)(1)/(2*(Pd|einf)**2)[0]

clifford.tools.g3c.project_points_to_plane

clifford.tools.g3c.**project_points_to_plane**(*point_list*, *plane*)
Takes a load of points and projects them onto a plane

clifford.tools.g3c.project_points_to_sphere

clifford.tools.g3c.**project_points_to_sphere**(*point_list*, *sphere*, *closest=True*)
Takes a load of points and projects them onto a sphere

clifford.tools.g3c.project_points_to_circle

clifford.tools.g3c.**project_points_to_circle**(*point_list*, *circle*, *closest=True*)
Takes a load of point and projects them onto a circle The closest flag determines if it should be the closest or furthest point on the circle

clifford.tools.g3c.project_points_to_line

clifford.tools.g3c.**project_points_to_line**(*point_list*, *line*)
Takes a load of points and projects them onto a line

clifford.tools.g3c.iterative_closest_points_on_circles

clifford.tools.g3c.**iterative_closest_points_on_circles**(*C1*, *C2*, *niterations=20*)
Given two circles C1 and C2 this calculates the closest points on each of them to the other

Changed in version 1.3: Renamed from `closest_points_on_circles`

clifford.tools.g3c.closest_point_on_line_from_circle

clifford.tools.g3c.**closest_point_on_line_from_circle**(*C*, *L*, *eps=1e-06*)

Returns the point on the line L that is closest to the circle C Uses the algorithm described in Appendix A of Andreas Aristidou's PhD thesis

New in version 1.3.

clifford.tools.g3c.closest_point_on_circle_from_line

clifford.tools.g3c.**closest_point_on_circle_from_line**(*C*, *L*, *eps=1e-06*)

Returns the point on the circle C that is closest to the line L Uses the algorithm described in Appendix A of Andreas Aristidou's PhD thesis

New in version 1.3.

clifford.tools.g3c.iterative_closest_points_circle_line

clifford.tools.g3c.**iterative_closest_points_circle_line**(*C*, *L*, *niterations=20*)

Given a circle C and line L this calculates the closest points on each of them to the other.

This is an iterative algorithm based on heuristics Nonetheless it appears to give results on par with *closest_point_on_circle_from_line()*.

Changed in version 1.3: Renamed from `closest_points_circle_line`

clifford.tools.g3c.iterative_furthest_points_on_circles

clifford.tools.g3c.**iterative_furthest_points_on_circles**(*C1*, *C2*, *niterations=20*)

Given two circles C1 and C2 this calculates the closest points on each of them to the other

Changed in version 1.3: Renamed from `furthest_points_on_circles`

clifford.tools.g3c.sphere_beyond_plane

clifford.tools.g3c.**sphere_beyond_plane**(*sphere*, *plane*)

Check if the sphere is fully beyond the plane in the direction of the plane normal

clifford.tools.g3c.sphere_behind_plane

clifford.tools.g3c.**sphere_behind_plane**(*sphere*, *plane*)

Check if the sphere is fully behind the plane in the direction of the plane normal, ie the opposite of sphere_beyond_plane

Misc

meet_val(a_val, b_val)	The meet algorithm as described in "A Covariant Approach to Geometry" I5*((I5*A) ^ (I5*B))
meet(A, B)	The meet algorithm as described in "A Covariant Approach to Geometry" I5*((I5*A) ^ (I5*B))
normalise_n_minus_1(mv)	Normalises a conformal point so that it has an inner product of -1 with einf
val_normalise_n_minus_1(mv_val)	Normalises a conformal point so that it has an inner product of -1 with einf
val_apply_rotor(mv_val, rotor_val)	Applies rotor to multivector in a fast way - JITTED
apply_rotor(mv_in, rotor)	Applies rotor to multivector in a fast way
val_apply_rotor_inv(mv_val, rotor_val, ...)	Applies rotor to multivector in a fast way takes pre computed adjoint
apply_rotor_inv(mv_in, rotor, rotor_inv)	Applies rotor to multivector in a fast way takes pre computed adjoint
euc_dist(conf_mv_a, conf_mv_b)	Returns the distance between two conformal points
mult_with_ninf(mv)	Convenience function for multiplication with ninf
val_norm(mv_val)	Returns sqrt(abs(~A*A))
norm(mv)	Returns sqrt(abs(~A*A))
val_normalised(mv_val)	Returns A/sqrt(abs(~A*A))
normalised(mv)	fast version of the normal() function
val_up(mv_val)	Fast jitted up mapping
fast_up(mv)	Fast up mapping
val_normalInv(mv_val)	A fast, jitted version of normalInv
val_homo(mv_val)	A fast, jitted version of homo()
val_down(mv_val)	A fast, jitted version of down()
fast_down(mv)	A fast version of down()
dual_func(a_val)	Fast dual
fast_dual(a)	Fast dual
disturb_object(mv_object[, ...])	Disturbs an object by a random rotor
project_val(val, grade)	fast grade projection
get_line_reflection_matrix(lines[, n_power])	Generates the matrix that sums the reflection of a point in many lines
val_get_line_reflection_matrix(line_array, ...)	Generates the matrix that sums the reflection of a point in many lines
val_truncated_get_line_reflection_matrix(...)	Generates the truncated matrix that sums the reflection of a point in many lines
interpret_multivector_as_object(mv)	Takes an input multivector and returns what kind of object it is
normalise_TR_to_unit_T(TR)	Takes in a TR rotor extracts the R and T normalises the T to unit displacement magnitude rebuilds the TR rotor with the new displacement rotor returns the new TR and the original length of the T rotor
scale_TR_translation(TR, scale)	Takes in a TR rotor and a scale extracts the R and T scales the T displacement magnitude by scale rebuilds the TR rotor with the new displacement rotor returns the new TR rotor
val_unsign_sphere(S)	Normalises the sign of a sphere

clifford.tools.g3c.meet_val

clifford.tools.g3c.**meet_val**(*a_val*, *b_val*)
The meet algorithm as described in "A Covariant Approach to Geometry" I5*((I5*A) ^ (I5*B))

clifford.tools.g3c.meet

clifford.tools.g3c.**meet**(*A*, *B*)
The meet algorithm as described in "A Covariant Approach to Geometry" I5*((I5*A) ^ (I5*B))

clifford.tools.g3c.normalise_n_minus_1

clifford.tools.g3c.**normalise_n_minus_1**(*mv*)
Normalises a conformal point so that it has an inner product of -1 with einf

clifford.tools.g3c.val_normalise_n_minus_1

clifford.tools.g3c.**val_normalise_n_minus_1**(*mv_val*)
Normalises a conformal point so that it has an inner product of -1 with einf

clifford.tools.g3c.val_apply_rotor

clifford.tools.g3c.**val_apply_rotor**(*mv_val*, *rotor_val*)
Applies rotor to multivector in a fast way - JITTED

clifford.tools.g3c.apply_rotor

clifford.tools.g3c.**apply_rotor**(*mv_in*, *rotor*)
Applies rotor to multivector in a fast way

clifford.tools.g3c.val_apply_rotor_inv

clifford.tools.g3c.**val_apply_rotor_inv**(*mv_val*, *rotor_val*, *rotor_val_inv*)
Applies rotor to multivector in a fast way takes pre computed adjoint

clifford.tools.g3c.apply_rotor_inv

clifford.tools.g3c.**apply_rotor_inv**(*mv_in*, *rotor*, *rotor_inv*)
Applies rotor to multivector in a fast way takes pre computed adjoint

clifford.tools.g3c.euc_dist

clifford.tools.g3c.**euc_dist**(*conf_mv_a*, *conf_mv_b*)
Returns the distance between two conformal points

clifford.tools.g3c.mult_with_ninf

clifford.tools.g3c.**mult_with_ninf**(*mv*)
Convenience function for multiplication with ninf

clifford.tools.g3c.val_norm

clifford.tools.g3c.**val_norm**(*mv_val*)
Returns sqrt(abs(~A*A))

clifford.tools.g3c.norm

clifford.tools.g3c.**norm**(*mv*)
Returns sqrt(abs(~A*A))

clifford.tools.g3c.val_normalised

clifford.tools.g3c.**val_normalised**(*mv_val*)
Returns A/sqrt(abs(~A*A))

clifford.tools.g3c.normalised

clifford.tools.g3c.**normalised**(*mv*)
fast version of the normal() function

clifford.tools.g3c.val_up

clifford.tools.g3c.**val_up**(*mv_val*)
Fast jitted up mapping

clifford.tools.g3c.fast_up

clifford.tools.g3c.**fast_up**(*mv*)
Fast up mapping

clifford.tools.g3c.val_normalInv

clifford.tools.g3c.**val_normalInv**(*mv_val*)
A fast, jitted version of normalInv

clifford.tools.g3c.val_homo

clifford.tools.g3c.**val_homo**(*mv_val*)
A fast, jitted version of homo()

clifford.tools.g3c.val_down

clifford.tools.g3c.**val_down**(*mv_val*)
A fast, jitted version of down()

clifford.tools.g3c.fast_down

clifford.tools.g3c.**fast_down**(*mv*)
A fast version of down()

clifford.tools.g3c.dual_func

clifford.tools.g3c.**dual_func**(*a_val*)
Fast dual

clifford.tools.g3c.fast_dual

clifford.tools.g3c.**fast_dual**(*a*)
Fast dual

clifford.tools.g3c.disturb_object

clifford.tools.g3c.**disturb_object**(*mv_object*, *maximum_translation=0.01*, *maximum_angle=0.01*)
Disturbs an object by a random rotor

clifford.tools.g3c.project_val

clifford.tools.g3c.**project_val**(*val*, *grade*)
fast grade projection

clifford.tools.g3c.get_line_reflection_matrix

clifford.tools.g3c.**get_line_reflection_matrix**(*lines*, *n_power=1*)

Generates the matrix that sums the reflection of a point in many lines

clifford.tools.g3c.val_get_line_reflection_matrix

clifford.tools.g3c.**val_get_line_reflection_matrix**(*line_array: numpy.ndarray*, *n_power: int*) → numpy.ndarray

Generates the matrix that sums the reflection of a point in many lines

clifford.tools.g3c.val_truncated_get_line_reflection_matrix

clifford.tools.g3c.**val_truncated_get_line_reflection_matrix**(*line_array: numpy.ndarray*, *n_power: int*) → numpy.ndarray

Generates the truncated matrix that sums the reflection of a point in many lines

clifford.tools.g3c.interpret_multivector_as_object

clifford.tools.g3c.**interpret_multivector_as_object**(*mv*)

Takes an input multivector and returns what kind of object it is

-1 -> not a blade 0 -> a 1 vector but not a point 1 -> a euclidean point 2 -> a conformal point 3 -> a point pair 4 -> a circle 5 -> a line 6 -> a sphere 7 -> a plane

Similar to *clifford.tools.classify.classify()*, although that function does a little more work in order to produce full characterizations.

clifford.tools.g3c.normalise_TR_to_unit_T

clifford.tools.g3c.**normalise_TR_to_unit_T**(*TR*)

Takes in a TR rotor extracts the R and T normalises the T to unit displacement magnitude rebuilds the TR rotor with the new displacement rotor returns the new TR and the original length of the T rotor

clifford.tools.g3c.scale_TR_translation

clifford.tools.g3c.**scale_TR_translation**(*TR*, *scale*)

Takes in a TR rotor and a scale extracts the R and T scales the T displacement magnitude by scale rebuilds the TR rotor with the new displacement rotor returns the new TR rotor

clifford.tools.g3c.val_unsign_sphere

clifford.tools.g3c.**val_unsign_sphere**(*S*)
Normalises the sign of a sphere

Root Finding

dorst_norm_val(sigma_val)	Square Root of Rotors - Implements the norm of a rotor
check_sigma_for_positive_root_val(sigma_val)	Square Root of Rotors - Checks for a positive root
check_sigma_for_positive_root(sigma)	Square Root of Rotors - Checks for a positive root
check_sigma_for_negative_root_val(sigma_value)	Square Root of Rotors - Checks for a negative root
check_sigma_for_negative_root(sigma)	Square Root of Rotors - Checks for a negative root
check_infinite_roots_val(sigma_value)	Square Root of Rotors - Checks for a infinite roots
check_infinite_roots(sigma)	Square Root of Rotors - Checks for a infinite roots
positive_root_val(sigma_val)	Square Root of Rotors - Evaluates the positive root Square Root of Rotors - Evaluates the positive root
negative_root_val(sigma_val)	Square Root of Rotors - Evaluates the positive root
positive_root(sigma)	Square Root of Rotors - Evaluates the positive root
negative_root(sigma)	Square Root of Rotors - Evaluates the negative root
general_root_val(sigma_value)	Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg
general_root(sigma)	The general case of the root of a grade 0, 4 multivector
val_annihilate_k(K_val, C_val)	Removes K from C = KX via (K[0] - K[4])*C
annihilate_k(K, C)	Removes K from C = KX via (K[0] - K[4])*C
pos_twiddle_root_val(C_value)	Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg
neg_twiddle_root_val(C_value)	Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg
pos_twiddle_root(C)	Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg
neg_twiddle_root(C)	Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg
square_roots_of_rotor(R)	Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg
n_th_rotor_root(R, n)	Takes the n_th root of rotor R n must be a power of 2
interp_objects_root(C1, C2, alpha)	Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly linearly interpolates conformal objects Return a valid object from the addition result C
general_object_interpolation(... [, kind])	Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 This is a general interpolation through the

continues on next page

Table 27 – continued from previous page

average_objects(obj_list[, weights, ...])	Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly averages conformal objects Return a valid object from the addition result C
val_average_objects_with_weights(obj_array, ...)	Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly averages conformal objects Return a valid object from the addition result C
val_average_objects(obj_array)	Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly averages conformal objects Return a valid object from the addition result C
rotor_between_objects(X1, X2)	Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a rotor that takes X1 to X2 Return a valid object from the addition result 1 + gamma*X2X1
val_rotor_between_objects_root(X1, X2)	Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a rotor that takes X1 to X2 Uses the square root of rotors for efficiency and numerical stability
val_rotor_between_objects_explicit(X1, X2)	Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a rotor that takes X1 to X2
calculate_S_over_mu(X1, X2)	Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a factor that corrects the X1 + X2 back to a blade
val_rotor_between_lines(L1_val, L2_val)	Implements a very optimised rotor line to line extraction
rotor_between_lines(L1, L2)	return the rotor between two lines
rotor_between_planes(P1, P2)	return the rotor between two planes
val_rotor_rotor_between_planes(P1_val, P2_val)	return the rotor between two planes

clifford.tools.g3c.dorst_norm_val

clifford.tools.g3c.**dorst_norm_val**(*sigma_val*)

Square Root of Rotors - Implements the norm of a rotor

clifford.tools.g3c.check_sigma_for_positive_root_val

clifford.tools.g3c.**check_sigma_for_positive_root_val**(*sigma_val*)
Square Root of Rotors - Checks for a positive root

clifford.tools.g3c.check_sigma_for_positive_root

clifford.tools.g3c.**check_sigma_for_positive_root**(*sigma*)
Square Root of Rotors - Checks for a positive root

clifford.tools.g3c.check_sigma_for_negative_root_val

clifford.tools.g3c.**check_sigma_for_negative_root_val**(*sigma_value*)
Square Root of Rotors - Checks for a negative root

clifford.tools.g3c.check_sigma_for_negative_root

clifford.tools.g3c.**check_sigma_for_negative_root**(*sigma*)
Square Root of Rotors - Checks for a negative root

clifford.tools.g3c.check_infinite_roots_val

clifford.tools.g3c.**check_infinite_roots_val**(*sigma_value*)
Square Root of Rotors - Checks for a infinite roots

clifford.tools.g3c.check_infinite_roots

clifford.tools.g3c.**check_infinite_roots**(*sigma*)
Square Root of Rotors - Checks for a infinite roots

clifford.tools.g3c.positive_root_val

clifford.tools.g3c.**positive_root_val**(*sigma_val*)
Square Root of Rotors - Evaluates the positive root Square Root of Rotors - Evaluates the positive root

clifford.tools.g3c.negative_root_val

clifford.tools.g3c.**negative_root_val**(*sigma_val*)
Square Root of Rotors - Evaluates the positive root

clifford.tools.g3c.positive_root

clifford.tools.g3c.**positive_root**(*sigma*)
Square Root of Rotors - Evaluates the positive root

clifford.tools.g3c.negative_root

clifford.tools.g3c.**negative_root**(*sigma*)
Square Root of Rotors - Evaluates the negative root

clifford.tools.g3c.general_root_val

clifford.tools.g3c.**general_root_val**(*sigma_value*)
Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg

clifford.tools.g3c.general_root

clifford.tools.g3c.**general_root**(*sigma*)
The general case of the root of a grade 0, 4 multivector

clifford.tools.g3c.val_annihilate_k

clifford.tools.g3c.**val_annihilate_k**(*K_val*, *C_val*)
Removes K from C = KX via (K[0] - K[4])*C

clifford.tools.g3c.annihilate_k

clifford.tools.g3c.**annihilate_k**(*K*, *C*)
Removes K from C = KX via (K[0] - K[4])*C

clifford.tools.g3c.pos_twiddle_root_val

clifford.tools.g3c.**pos_twiddle_root_val**(*C_value*)
Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg

clifford.tools.g3c.neg_twiddle_root_val

clifford.tools.g3c.**neg_twiddle_root_val**(*C_value*)
Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg

clifford.tools.g3c.pos_twiddle_root

clifford.tools.g3c.**pos_twiddle_root**(*C*)

Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg

clifford.tools.g3c.neg_twiddle_root

clifford.tools.g3c.**neg_twiddle_root**(*C*)

Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg

clifford.tools.g3c.square_roots_of_rotor

clifford.tools.g3c.**square_roots_of_rotor**(*R*)

Square Root and Logarithm of Rotors in 3D Conformal Geometric Algebra Using Polar Decomposition Leo Dorst and Robert Valkenburg

clifford.tools.g3c.n_th_rotor_root

clifford.tools.g3c.**n_th_rotor_root**(*R*, *n*)

Takes the n_th root of rotor R n must be a power of 2

clifford.tools.g3c.interp_objects_root

clifford.tools.g3c.**interp_objects_root**(*C1*, *C2*, *alpha*)

Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly linearly interpolates conformal objects Return a valid object from the addition result C

clifford.tools.g3c.general_object_interpolation

clifford.tools.g3c.**general_object_interpolation**(*object_alpha_array*, *object_list*, *new_alpha_array*, *kind='linear'*)

Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 This is a general interpolation through the

clifford.tools.g3c.average_objects

clifford.tools.g3c.**average_objects**(*obj_list*, *weights=[]*, *check_grades=True*)

Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly averages conformal objects Return a valid object from the addition result C

clifford.tools.g3c.val_average_objects_with_weights

clifford.tools.g3c.**val_average_objects_with_weights**(*obj_array*, *weights_array*)
Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly averages conformal objects Return a valid object from the addition result C

clifford.tools.g3c.val_average_objects

clifford.tools.g3c.**val_average_objects**(*obj_array*)
Hadfield and Lasenby, Direct Linear Interpolation of Geometric Objects, AGACSE2018 Directly averages conformal objects Return a valid object from the addition result C

clifford.tools.g3c.rotor_between_objects

clifford.tools.g3c.**rotor_between_objects**(*X1*, *X2*)
Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a rotor that takes X1 to X2 Return a valid object from the addition result 1 + gamma*X2X1

clifford.tools.g3c.val_rotor_between_objects_root

clifford.tools.g3c.**val_rotor_between_objects_root**(*X1*, *X2*)
Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a rotor that takes X1 to X2 Uses the square root of rotors for efficiency and numerical stability

clifford.tools.g3c.val_rotor_between_objects_explicit

clifford.tools.g3c.**val_rotor_between_objects_explicit**(*X1*, *X2*)
Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a rotor that takes X1 to X2

Implements an optimised version of:

```
gamma1 = (X1 * X1)[0]
gamma2 = (X2 * X2)[0]

M12 = X1*X2 + X2*X1
K = 2 + gamma1*M12

if np.sum(np.abs(K.value)) < 0.0000001:
    return 1 + 0*e1

if sum(np.abs(M12(4).value)) > 0.0000001:
    lamb = (-(K(4) * K(4)))[0]
    mu = K[0]**2 + lamb
    root_mu = np.sqrt(mu)
    if abs(lamb) < 0.0000001:
        beta = 1.0/(2*np.sqrt(K[0]))
    else:
        beta_sqrd = 1/(2*(root_mu + K[0]))
        beta = np.sqrt(beta_sqrd)
    R = ( beta*K(4) - (1/(2*beta)) )*(1 + gamma1*X2*X1)/(root_mu)
```

(continues on next page)

(continued from previous page)

```
    return R
else:
    return (1 + gamma1*X2*X1)/(np.sqrt(abs(K[0])))
```

clifford.tools.g3c.calculate_S_over_mu

clifford.tools.g3c.**calculate_S_over_mu**(*X1*, *X2*)
Lasenby and Hadfield AGACSE2018 For any two conformal objects X1 and X2 this returns a factor that corrects the X1 + X2 back to a blade

clifford.tools.g3c.val_rotor_between_lines

clifford.tools.g3c.**val_rotor_between_lines**(*L1_val*, *L2_val*)
Implements a very optimised rotor line to line extraction

clifford.tools.g3c.rotor_between_lines

clifford.tools.g3c.**rotor_between_lines**(*L1*, *L2*)
return the rotor between two lines

clifford.tools.g3c.rotor_between_planes

clifford.tools.g3c.**rotor_between_planes**(*P1*, *P2*)
return the rotor between two planes

clifford.tools.g3c.val_rotor_rotor_between_planes

clifford.tools.g3c.**val_rotor_rotor_between_planes**(*P1_val*, *P2_val*)
return the rotor between two planes

Submodules

object_fitting	Tools for fitting geometric primitives to point clouds

clifford.tools.g3c.object_fitting

Tools for fitting geometric primitives to point clouds

Object Fitting

fit_circle(point_list)	Performs Leo Dorsts circle fitting technique
val_fit_circle(point_list)	Performs Leo Dorsts circle fitting technique
fit_line(point_list)	Does line fitting with combo J.Lasenbys method and L.
val_fit_line(point_list)	Does line fitting with combo J.Lasenbys method and L.
fit_sphere(point_list)	Performs Leo Dorsts sphere fitting technique
val_fit_sphere(point_list)	Performs Leo Dorsts sphere fitting technique
fit_plane(point_list)	Does plane fitting with combo J.Lasenbys method and L.
val_fit_plane(point_list)	Does plane fitting with combo J.Lasenbys method and L.

clifford.tools.g3c.object_fitting.fit_circle

clifford.tools.g3c.object_fitting.**fit_circle**(*point_list*)
Performs Leo Dorsts circle fitting technique

clifford.tools.g3c.object_fitting.val_fit_circle

clifford.tools.g3c.object_fitting.**val_fit_circle**(*point_list*)
Performs Leo Dorsts circle fitting technique

clifford.tools.g3c.object_fitting.fit_line

clifford.tools.g3c.object_fitting.**fit_line**(*point_list*)
Does line fitting with combo J.Lasenbys method and L. Dorsts

clifford.tools.g3c.object_fitting.val_fit_line

clifford.tools.g3c.object_fitting.**val_fit_line**(*point_list*)
Does line fitting with combo J.Lasenbys method and L. Dorsts

clifford.tools.g3c.object_fitting.fit_sphere

clifford.tools.g3c.object_fitting.**fit_sphere**(*point_list*)
Performs Leo Dorsts sphere fitting technique

clifford.tools.g3c.object_fitting.val_fit_sphere

clifford.tools.g3c.object_fitting.**val_fit_sphere**(*point_list*)
Performs Leo Dorsts sphere fitting technique

clifford.tools.g3c.object_fitting.fit_plane

clifford.tools.g3c.object_fitting.**fit_plane**(*point_list*)
Does plane fitting with combo J.Lasenbys method and L. Dorsts

clifford.tools.g3c.object_fitting.val_fit_plane

clifford.tools.g3c.object_fitting.**val_fit_plane**(*point_list*)
Does plane fitting with combo J.Lasenbys method and L. Dorsts

1.3.2 Classifying conformal GAs

classify	Tools for interpreting conformal blades

clifford.tools.classify

Tools for interpreting conformal blades

clifford.tools.classify.**classify**(*x*) → *clifford.tools.classify.Blade*
Classify a conformal multivector into a parameterized geometric description.

The multivector should be from a *ConformalLayout*, such as the one returned by *clifford.conformalize()*.

Implemented based on the approach described in table 14.1 of *Geometric Algebra for Computer Science (Revised Edition)*.

Example usage:

```
>>> from clifford.g3c import *

>>> classify(e1)
DualFlat[1](flat=Plane(direction=-(1.0^e23), location=0))

>>> classify(einf)
InfinitePoint(direction=1.0)

>>> classify(up(e1))
Point(direction=1.0, location=(1.0^e1))
```

(continues on next page)

(continued from previous page)

```
>>> classify(up(3*e1)^up(4*e2))
PointPair(direction=-(3.0^e1) + (4.0^e2), location=(1.5^e1) + (2.0^e2), radius=2.
↪5)

>>> classify(up(e1)^up(e2)^up(e1+2*e2))
Circle(direction=-(2.0^e12), location=(1.0^e1) + (1.0^e2), radius=1.0)

>>> classify(up(e1)^up(e2)^up(e1+2*e2)^einf)
Plane(direction=-(2.0^e12), location=0)

>>> classify(up(e1)^e2)
Tangent[2](direction=(1.0^e2), location=(1.0^e1))

# how the inheritance works
>>> Point.mro()
[Point, Tangent[1], Round[1], Blade[1], Tangent, Round, Blade, object]
```

The reverse of this operation is *Blade.mv*.

The return type is a *Blade*:

class clifford.tools.classify.**Blade**(*layout*)
Base class for providing interpretation of blades.

Note that thanks to the unual metaclass, this class and its subclasses are have grade-specific specializations, eg `Blade[2]` is a type for blades of grade 2.

layout
The layout to which this blade belongs

Type *Layout*

property mv
Convert this back into its GA representation

The subclasses below are the four categories to which all blades belong, where E is a euclidean blade, and $T_p[X]$ represents a translation of the conformal blade X by the euclidean vector p.

class clifford.tools.classify.**Direction**(*direction*)
En_∞

direction
The euclidean direction, E

Type *MultiVector*

class clifford.tools.classify.**Flat**(*direction*, *location*)
$T_p[n_o \wedge (En_\infty)]$

direction
The euclidean direction, E

Type *MultiVector*

location
The closest point on this flat to the origin, p, as a euclidean vector.

Type *MultiVector*

class clifford.tools.classify.**DualFlat**(*flat*)
Dual of *Flat*

flat
The flat this is the dual of

Type *Flat*

class clifford.tools.classify.**Round**(*direction*, *location*, *radius*)
$T_p[(n_o + \frac{1}{2}\rho^2 n_\infty)E]$

direction
The euclidean direction, E

Type *MultiVector*

location
The euclidean center, p

Type *MultiVector*

radius
The radius, ρ, which may be imaginary

Type float or complex

These can be constructed directly, and will attempt to show a grade-specific interpretation:

```
>>> from clifford.g3c import *
>>> Round(location=e1, direction=e1^e2, radius=1)
Circle(direction=(1^e12), location=(1^e1), radius=1)
>>> Round(direction=e1^e2^e3, location=e1, radius=1)
Sphere(direction=(1^e123), location=(1^e1), radius=1)
```

Aliased types

In addition, aliases are created for specific grades of the above types, with more convenient names:

class clifford.tools.classify.**Tangent**(*direction*, *location*)
Bases: *clifford.tools.classify.Round*

A *Round* of radius 0, $T_p[n_o E]$

class clifford.tools.classify.**Point**(*direction*, *location*)
Bases: clifford.tools.classify.Tangent[1]

A conformal point, A

class clifford.tools.classify.**PointFlat**(*direction*, *location*)
Bases: clifford.tools.classify.Flat[2]

A flat point, $A \wedge n_\infty$

class clifford.tools.classify.**Line**(*direction*, *location*)
Bases: clifford.tools.classify.Flat[3]

A line, $A \wedge B \wedge n_\infty$

class clifford.tools.classify.**Plane**(*direction*, *location*)
Bases: clifford.tools.classify.Flat[4]

A line, $A \wedge B \wedge C \wedge n_\infty$

class clifford.tools.classify.**PointPair**(*direction*, *location*, *radius*)
Bases: clifford.tools.classify.Round[2]

A point pair, $A \wedge B$

class clifford.tools.classify.**Circle**(*direction*, *location*, *radius*)
Bases: clifford.tools.classify.Round[3]

A circle, $A \wedge B \wedge C$

class clifford.tools.classify.**Sphere**(*direction*, *location*, *radius*)
Bases: clifford.tools.classify.Round[4]

A sphere, $A \wedge B \wedge C \wedge D$

class clifford.tools.classify.**InfinitePoint**(*direction*)
Bases: clifford.tools.classify.Direction[1]

A scalar multiple of n_∞

1.3.3 Determining Rotors From Frame Pairs or Orthogonal Matrices

Given two frames that are related by a orthogonal transform, we seek a rotor which enacts the transform. Details of the mathematics and psuedo-code used the create the algorithms below can be found at Allan Cortzen's website.

http://ctz.dk/geometric-algebra/frames-to-versor-algorithm/

There are also some helper functions which can be used to translate matrices into GA frames, so an orthogonal (or complex unitary) matrix can be directly translated into a Versor.

orthoFrames2Versor(B[, A, delta, eps, det, ...])	Determines versor for two frames related by an orthogonal transform
orthoMat2Versor(A[, eps, layout, is_complex])	Translates an orthogonal (or unitary) matrix to a Versor
mat2Frame(A[, layout, is_complex])	Translates a (possibly complex) matrix into a real vector frame

clifford.tools.orthoFrames2Versor

clifford.tools.**orthoFrames2Versor**(*B*, *A=None*, *delta=0.001*, *eps=None*, *det=None*, *remove_scaling=False*)
Determines versor for two frames related by an orthogonal transform

Based on [1,2]. This works in Euclidean spaces and, under special circumstances in other signatures. see [1] for limitaions/details

Parameters

- **B** (*list of vectors, or clifford.Frame*) – the set of vectors after the transform, and homogenzation. ie B=(B/Bleinf)
- **A** (*list of vectors, or clifford.Frame*) – the set of vectors before the transform. If *None* we assume A is the basis given B.layout.basis_vectors_lst
- **delta** (*float*) – Tolerance for reflection/rotation determination. If the normalized distance between A[i] and B[i] is larger than delta, we use reflection, otherwise use rotation.
- **eps** (*float*) – Tolerance on spinor determination. if pseudoscalar of A differs in magnitude from pseudoscalar of B by eps, then we have spinor. If *None*, use the *clifford.eps()* global eps.
- **det** (*[+1, -1, None]*) – The sign of the determinant of the versor, if known. If it is known a-priori that the versor is a rotation vs a reflection, this fact might be needed to correctly append an additional reflection which leaves transformed points invariant. See 4.6.3 [2].

- **remove_scaling** (*Bool*) – Remove the effects of homogenzation from frame B. This is needed if you are working in CGA, but the input data is given in the original space. See *omoh* method for more. See 4.6.2 of [2]

Returns

- **R** (*clifford.Multivector*) – the Versor.
- **rs** (*list of clifford.Multivectors*) – ordered list of found reflectors/rotors.

References

[1] http://ctz.dk/geometric-algebra/frames-to-versor-algorithm/

[2] Reconstructing Rotations and Rigid Body Motions from Exact Point Correspondences Through Reflections, Daniel Fontijne and Leo Dorst

clifford.tools.orthoMat2Versor

clifford.tools.**orthoMat2Versor**(*A*, *eps=None*, *layout=None*, *is_complex=None*)

Translates an orthogonal (or unitary) matrix to a Versor

A is interpreted as the frame produced by transforming a orthonormal frame by an orthogonal transform. Given this relation, this function will find the versor which enacts this transform.

clifford.tools.mat2Frame

clifford.tools.**mat2Frame**(*A*, *layout=None*, *is_complex=None*)

Translates a (possibly complex) matrix into a real vector frame

The rows and columns are interpreted as follows

- M,N = shape(A)
- M = dimension of space
- N = number of vectors

If A is complex M and N are doubled.

Parameters **A** (*ndarray*) – MxN matrix representing vectors

1.4 operator functions (clifford.operator)

This module exists to enable functional programming via `functools.reduce()`. It can be thought of as equivalent to the builtin `operator` module, but for the operators from geometric algebra.

```
>>> import functools
>>> import clifford.operator
>>> from clifford.g3 import *
>>> Ms = [e1, e1 + e2, e2 + e3]  # list of multivectors
>>> assert functools.reduce(clifford.operator.op, Ms) == Ms[0] ^ Ms[1] ^ Ms[2]
```

clifford.operator.**gp**(*M*, *N*)

Geometric product function MN, equivalent to `M * N`.

M and N must be from the same layout

clifford.operator.**op**(*M*, *N*)
Outer product function $M \wedge N$, equivalent to `M ^ N`.

M and N must be from the same layout

clifford.operator.**ip**(*M*, *N*)
Hestenes inner product function $M \bullet N$, equivalent to `M | N`.

M and N must be from the same layout

Changed in version 1.3.0: These functions used to be in *clifford*, but have been moved to this submodule.

1.5 transformations (clifford.transformations)

New in version 1.3.0.

This module may in future become the home for optimized rotor transformations, or non-linear transformations.

See the *Linear transformations* tutorial for an introduction to how to use parts of this module.

1.5.1 Base classes

clifford.transformations.**Transformation = typing.Callable[[clifford._multivector.MultiVecto**
A callable mapping one MultiVector to another.

class clifford.transformations.**Linear**
A transformation which is linear, such that for scalar a_i, $f(a_1 x_1 + a_2 x_2) = a_1 f(x_1) + a_2 f(x_2)$.

class clifford.transformations.**FixedLayout**(*layout_src: clifford._layout.Layout*, *layout_dst: clifford._layout.Layout = None*)
A transformation with a fixed source and destination layout.

Parameters

- **layout_src** (*Layout* of S dimensions) – The layout from which this transformation takes multivectors as input
- **layout_dst** (*Layout* of D dimensions) – The layout in which this transformation produces multivectors as output. Defaults to the same as the input.

1.5.2 Matrix-backed implementations

class clifford.transformations.**LinearMatrix**(*matrix*, *layout_src: clifford._layout.Layout*, *layout_dst: clifford._layout.Layout = None*)
Linear transformation implemented by a matrix

Transformations need not be grade preserving.

Parameters

- **matrix** (*(2**D, 2**S) array_like*) – A matrix that transforms multivectors from *layout_src* with 2^S elements to multivectors in *layout_dst* with 2^D elements, by left-multiplication.
- **layout_src** (*Layout of S dimensions*) – Passed on to *FixedLayout*.
- **layout_dst** (*Layout of D dimensions*) – Passed on to *FixedLayout*.

See also:

clifford.BladeMap A faster but less general approach that works on basis blades

property adjoint
The adjoint transformation

classmethod from_function(*func: Callable[[clifford._multivector.MultiVector], clifford._multivector.MultiVector]*, *layout_src: clifford._layout.Layout*, *layout_dst: clifford._layout.Layout = None*) → *clifford.transformations.LinearMatrix*
Build a linear transformation from the result of a function applied to each basis blade.

Parameters

- **func** – A function taking basis blades from *layout_src* that produces multivectors in *layout_dst*.
- **layout_src** (*Layout of S dimensions*) – The layout to pass into the generating function
- **layout_dst** (*Layout of D dimensions*) – The layout the generating function is expected to produce. If not passed, this is inferred.

Example

```
>>> from clifford import transformations, Layout
>>> l = Layout([1, 1])
>>> e1, e2 = l.basis_vectors_lst
>>> rot_90 = transformations.LinearMatrix.from_function(lambda x: (1 +␣
↪e1*e2)*x*(1 - e1*e2)/2, l)
>>> rot_90(e1)
(1.0^e2)
>>> rot_90(e2)
-(1.0^e1)
>>> rot_90(e1*e2)
(1.0^e12)
```

See also:

LinearMatrix.from_rotor() a shorter way to spell the above example

clifford.linear_operator_as_matrix() a lower-level function for working with a subset of basis blades

classmethod from_rotor(*rotor: clifford._multivector.MultiVector*) → *clifford.transformations.LinearMatrix*
Build a linear transformation from the result of applying a rotor sandwich.

The resulting transformation operates within the algebra of the provided rotor.

Parameters **rotor** – The rotor to apply

Example

```
>>> from clifford import transformations, Layout
>>> l = Layout([1, 1])
>>> e1, e2 = l.basis_vectors_lst
>>> rot_90 = transformations.LinearMatrix.from_rotor(1 + e1*e2)
>>> rot_90(e1)
(1.0^e2)
>>> rot_90(e2)
-(1.0^e1)
>>> rot_90(e1*e2)
(1.0^e12)
```

class clifford.transformations.**OutermorphismMatrix**(*matrix*, *layout_src: clifford._layout.Layout*, *layout_dst: clifford._layout.Layout = None*)

A generalization of a linear transformation to vectors via the outer product.

Namely, given a linear transformation $F(u) \to v$, this generalizes to the blades by outermorphism, $F(u_1 \wedge u_2) \to F(u_1) \wedge F(u_2)$, and to the multivectors by distributivity.

Such a transformation is grade preserving.

See GA4CS Chapter 4 for more information

Parameters

- **matrix** (*(D, S) array_like*) – A matrix that transforms vectors from *layout_src* of size S to vectors in *layout_dst* of size D by left-multiplication.
- **layout_src** (*Layout of S dimensions*) – Passed on to *FixedLayout*.
- **layout_dst** (*Layout of D dimensions*) – Passed on to *FixedLayout*.

Example

We can construct a simple transformation that permutes and non-uniformly scales the basis vectors:

```
>>> from clifford import transformations, Layout
>>> layout = Layout([1, 1, 1])
>>> e1, e2, e3 = layout.basis_vectors_lst
>>> layout_new = Layout([1, 1, 1], names='f')
>>> m = np.array([[0, 1, 0],
...               [0, 0, 2],
...               [3, 0, 0]])
>>> lt = transformations.OutermorphismMatrix(m, layout, layout_new)
```

Applying it to some multivectors:

```
>>> # the transformation we specified
>>> lt(e1), lt(e2), lt(e3)
((3^f3), (1^f1), (2^f2))

>>> # the one deduced by outermorphism
>>> lt(e1^e2), lt(e2^e3), lt(e1^e3)
(-(3^f13), (2^f12), -(6^f23))

>>> # and by distributivity
```

(continues on next page)

(continued from previous page)

```
>>> lt(1 + (e1^e2^e3))
1 + (6^f123)
```

1.5.3 Helper functions

clifford.transformations.**between_basis_vectors**(*layout_src: clifford._layout.Layout*, *layout_dst: clifford._layout.Layout*, *mapping: Dict[Any, Any] = None*) → *clifford.transformations.OutermorphismMatrix*

Construct an outermorphism that maps basis vectors from one layout to basis vectors in another.

Parameters

- **layout_src** (Layout) – Passed on to *FixedLayout*.
- **layout_dst** (Layout) – Passed on to *FixedLayout*.
- **mapping** – If provided, a dictionary mapping the ids of source basis vectors to the ids of destination basis vectors. For example, `{1: 2, 2: 3, 3: 1}` would permute the basis vectors of *clifford.g3*.

Example

See the tutorial on *Working with custom algebras* for a motivating example.

CHAPTER

TWO

PREDEFINED ALGEBRAS

The easiest way to get started with `clifford` is to use one of several predefined algebras:

- `g2`: 2D Euclidean, `Cl(2)`. See *Quick Start (G2)* for some examples.
- `g3`: 3D Euclidean, `Cl(3)`. See *The Algebra Of Space (G3)* for some examples.
- `g4`: 4D Euclidean, `Cl(4)`.
- `g2c`: Conformal space for G2, `Cl(3, 1)`. See *Conformal Geometric Algebra* for some examples.
- `g3c`: Conformal space for G3, `Cl(4, 1)`.
- `pga`: Projective space for G3 `Cl(3, 0, 1)`.
- `gac`: Geometric Algebra for Conics, `Cl(5, 3)`.
- `dpga`: Double PGA also referred to as the Mother Algebra, `Cl(4, 4)`.
- `dg3c`: Double Conformal Geometric Algebra, effectively two g3c algebras glued together `Cl(8, 2)`.

By using the pre-defined algebras in place of calling `Cl` directly, you will often find that your program starts up faster.

clifford.<predefined>.e<ijk>
All of these modules expose the basis blades as attributes, and can be used like so

```
In [1]: from clifford import g2

In [2]: g2.e1 * g2.e2
Out[2]: (1^e12)
```

Additionally, they define the following attributes, which contain the return values of *clifford.Cl()*:

clifford.<predefined>.**layout**
The associated *clifford.Layout*

```
In [3]: g2.layout
Out[3]:
Layout([1, 1],
       ids=BasisVectorIds.ordered_integers(2),
       order=BasisBladeOrder.shortlex(2),
       names=['', 'e1', 'e2', 'e12'])
```

clifford.<predefined>.**blades**
A shorthand for `Layout.blades()`

```
In [4]: g2.blades
Out[4]: {'': 1, 'e1': (1^e1), 'e2': (1^e2), 'e12': (1^e12)}
```

For interactive use, it's very handy to use `import *`

```
In [5]: from clifford.g2 import *

In [6]: e1, e2, e12
Out[6]: ((1^e1), (1^e2), (1^e12))
```

For the conformal layouts *g2c* and *g3c*, the full contents of the `stuff` result of *clifford.conformalize()* is also exposed as members of the module.

CHAPTER

THREE

CHANGELOG

3.1 Changes in 1.3.x

- Python 3.8 is officially supported. 1.2.0 was pinned to a bad numba version that was incompatible with 3.8.
- A new *clifford.operator* module to contain the previously undocumented *gp()*, *op()*, and *ip()* helpers.
- A new *clifford.transformations* module for linear transformations.
- Two new *Predefined Algebras*, *clifford.dpga* and *clifford.dg3c*.
- Improvements throughout the documentation:
  - Better overall structure, visible in the docs sidebar.
  - New tutorials for *Conformal Geometric Algebra* on visualization and applications.
  - New tutorial on *Working with custom algebras*.
  - New tutorial on *Linear transformations*.
  - New links at the top of each notebook tutorial, to run examples from the browser.
- Faster algebra construction. `Cl(3)` is now 100× faster, and `Cl(6)` is 20× faster. This is achieved by deferring product JIT compilation until the product is used for the first time.
- Additional testing and assorted improvements for *clifford.tools.g3c*:
  - *closest_point_on_circle_from_line()* has now been implemented roughly following the procedure described in Appendix A of Andreas Aristidou's PhD thesis.
  - *closest_point_on_line_from_circle()* has now also been added, projecting the result of *closest_point_on_circle_from_line()* to the line.
- *clifford.ugly()* now results in less ugly output for *Predefined Algebras*.

3.1.1 Bugs fixed

- *MultiVector.meet()* no longer produces zero erroneously.
- `mv[e1 + e12]` now raises `ValueError`, rather than being interpreted as `mv[e1]`.
- *ip()* (the inner product) no longer performs the outer product.
- *Layout.parse_multivector()* now throws `SyntaxError` on invalid input, rather than silenltly producing nonsense.
- *Layout.parse_multivector()* supports basis vector names which do not start with e.

- In *clifford.tools.g3c*:
  - *val_midpoint_between_lines()* now handles the case that the two lines are touching.
  - *val_fit_circle()* now correctly selects the first and second eigenvalue regardless of order.
  - *sphere_beyond_plane()* now tested and correct.
  - *sphere_behind_plane()* now tested and correct.
  - *val_unsign_sphere()* is now jitted, as it should have been from the start.
  - *get_nearest_plane_point()* correctly returns the conformal point rather than the 3D point.

3.1.2 Compatibility notes

- `clifford.grades_present` is deprecated in favor of *MultiVector.grades()*, the latter of which now takes an `eps` argument.
- `del mv[i]` is no longer legal, the equivalent `mv[i] = 0` should be used instead.
- `Layout.dict_to_multivector` has been removed. It was accidentally broken in 1.0.5, so there is little point deprecating it.
- *Layout.basis_names()* now returns a `list` of `str`, rather than a numpy array of `bytes`. The result now matches the construction order, rather than being sorted alphabetically. The order of *Layout.metric()* has been adjusted for consistency.
- The `imt_prod_mask`, `omt_prod_mask`, and `lcmt_prod_mask` attributes of *Layout* objects have been removed, as these were an unnecessary intermediate computation that had no need to be public.
- Some functions in *clifford.tools.g3c* have been renamed:
  - `closest_points_on_circles` has been renamed to *iterative_closest_points_on_circles()*.
  - `closest_points_circle_line` has been renamed to *iterative_closest_points_circle_line()*.
  - `furthest_points_on_circles` has been renamed to *iterative_furthest_points_on_circles()*.
- While this release is compatible with `numba` version 0.49.0, it is recommended to use 0.48.0 which does not emit as many warnings. See the installation instructions for how to follow this guidance.

3.1.3 Patch releases

- 1.3.1: Added compatibility with `numba` version 0.50.0.

3.2 Changes in 1.2.x

- `layout.isconformal`, `layout.einf`, and `layout.eo`, which were added in 1.0.4, have been removed. The first can now be spelt `isinstance(layout, clifford.ConformalLayout)`, and the other properties now exist only on *ConformalLayout* objects.
- *MultiVector.left_complement()* has been added for consistency with *MultiVector.right_complement()*.
- A new *clifford.tools.classify* module has been added for classifying blades.
- *Layout* objects print slightly more cleanly in Jupyter notebooks.
- *Layout.scalar* is now integral rather than floating point

3.2.1 Bugs fixed

- `pow(mv, 0)` gives the right result
- `nan` is now printed correctly when it appears in multivectors. Previously it was hidden
- *`MultiVector.right_complement()`* no longer performs the left complement.
- *`MultiVector.vee()`* has been corrected to have the same sign as *`MultiVector.meet()`*

3.2.2 Compatibility notes

- *`Layout.scalar`* is now integral rather than floating point, to match *`Layout.pseudoScalar`*.

3.3 Changes in 1.1.x

- Restores `layout.gmt`, `Layout.omt`, `Layout.imt`, and `Layout.lcmt`. A few releases ago, these existed but were dense. For memory reasons, they were then removed entirely. They have now been reinstated as `sparse.COO` matrix objects, which behave much the same as the original dense arrays.
- `MultiVectors` preserve their data type in addition, subtraction, and products. This means that integers remain integers until combined with floats. Note that this means in principle integer overflow is possible, so working with floats is still recommended. This also adds support for floating point types of other precision, such as `np.float32`.
- `setup.py` is now configured such that `pip2 install clifford` will not attempt to download this version, since it does not work at all on python 2.
- Documentation now includes examples of `pyganja` visualizations.

3.3.1 Compatibility notes

- `Layout.blades()` now includes the scalar `1`, as do other similar functions.
- `MultiVector.grades()` now returns a `set` not a `list`. This means code like `mv.grades() == [0]` will need to change to `mv.grades() == {0}`, or to work both before and after this change, `set(mv.grades()) == {0}`.

3.3.2 Bugs fixed

- `mv[(i, j)]` would sometimes fail if the indices were not in canonical order.
- `mv == None` and `layout == None` would crash rather than return `False`.
- `blade.isVersor()` would return `False`.
- `layout.blades_of_grade(0)` would not return the list it claimed to return.

3.3.3 Internal changes

- Switch to `pytest` for testing.
- Enable code coverage.
- Split into smaller files.
- Remove python 2 compatibility code, which already no longer worked.

3.4 Changes 0.6-0.7

- Added a real license.
- Convert to NumPy instead of Numeric.

3.5 Changes 0.5-0.6

- `join()` and `meet()` actually work now, but have numerical accuracy problems
- added `clean()` to *`MultiVector`*
- added `leftInv()` and `rightInv()` to *`MultiVector`*
- moved `pseudoScalar()` and `invPS()` to *`MultiVector`* (so we can derive new classes from *`MultiVector`*)
- changed all of the instances of creating a new MultiVector to create an instance of `self.__class__` for proper inheritance
- fixed bug in laInv()
- fixed the massive confusion about how dot() works
- added left-contraction
- fixed embarrassing bug in gmt generation
- added `normal()` and `anticommutator()` methods
- fixed dumb bug in `elements()` that limited it to 4 dimensions

3.6 Acknowledgements

Konrad Hinsen fixed a few bugs in the conversion to numpy and adding some unit tests.

CHAPTER

FOUR

ISSUES

Warning: This document is kept for historic reasons, but may no longer reflect the current state of the latest release of `clifford`. For the most up to date source of issues, look at the GitHub issue tracker.

- Currently, algebras over 6 dimensions are very slow. this is because this module was written for *pedagogical* purposes. However, because the syntax for this module is so attractive, we plan to fix the performance problems, in the future...
- Due to Python's order of operations, the bit operators `^ << |` are evaluated after the normal arithmetic operators `+ - * /`, which do not follow the precedence expected in GA

```
# written        meaning            possibly intended
1^e1 + 2^e2   == 1^(e1+2)^e2     != (1^e0) + (2^e1)
e2 + e1|e2    == (e2 + e1)|e2    != e1 + (e1|e2)
```

This can also cause confusion within the bitwise operators:

```
# written        meaning            possibly intended
e1 << e2 ^ e1 == (e1 << e2) ^ e1 != e1 << (e2 ^ e1)
e1 ^ e2 | e1  == (e1 << e2) ^ e1 != e1 << (e2 ^ e1)
```

- Since `|` is the inner product and the inner product with a scalar vanishes by definition, an expression like:

```
(1|e0) + (2|e1)
```

is null. Use the outer product or full geometric product, to multiply scalars with *MultiVector*s. This can cause problems if one has code that mixes Python numbers and MultiVectors. If the code multiplies two values that can each be either type without checking, one can run into problems as `1 | 2` has a very different result from the same multiplication with scalar MultiVectors.

- Taking the inverse of a *MultiVector* will use a method proposed by Christian Perwass that involves the solution of a matrix equation. A description of that method follows:

  Representing multivectors as 2^{dims}-vectors (in the matrix sense), we can carry out the geometric product with a multiplication table. In pseudo-tensorish language (using summation notation)

  $$m_i g_{ijk} n_k = v_j$$

  Suppose m_i are known (M is the vector we are taking the inverse of), the g_{ijk} have been computed for this algebra, and $v_j = 1$ if the j'th element is the scalar element and 0 otherwise, we can compute the dot product $m_i g_{ijk}$. This yields a rank-2 matrix. We can then use well-established computational linear algebra techniques to solve this matrix equation for n_k. The `laInv` method does precisely that.

The usual, analytic, method for computing inverses ($M^{-1} = \tilde{M}/(M\tilde{M})$ iff $M\tilde{M} = |M|^2$) fails for those multivectors where `M*~M` is not a scalar. It is onl)y used if the `inv` method is manually set to point to `normalInv`.

My testing suggests that `laInv` works. In the cases where `normalInv` works, `laInv` returns the same result (within `_eps`). In all cases, `M * M.laInv() == 1.0` (within `_eps`). Use whichever you feel comfortable with.

Of course, a new issue arises with this method. The inverses found are sometimes dependant on the order of multiplication. That is:

```
M.laInv() * M == 1.0
M * M.laInv() != 1.0
```

XXX Thus, there are two other methods defined, `leftInv` and `rightInv` which point to `leftLaInv` and `rightLaInv`. The method `inv` points to `rightInv`. Should the user choose, `leftInv` and `rightInv` will both point to `normalInv`, which yields a left- and right-inverse that are the same should either exist (the proof is fairly simple).

- The basis vectors of any algebra will be orthonormal unless you supply your own multiplication tables (which you are free to do after the *`Layout`* constructor is called). A derived class could be made to calculate these tables for you (and include methods for generating reciprocal bases and the like).
- No care is taken to preserve the dtype of the arrays. The purpose of this module is pedagogical. If your application requires so many multivectors that storage becomes important, the class structure here is unsuitable for you anyways. Instead, use the algorithms from this module and implement application-specific data structures.
- Conversely, explicit typecasting is rare. *`MultiVector`*s will have integer coefficients if you instantiate them that way. Dividing them by Python integers will have the same consequences as normal integer division. Public outcry will convince me to add the explicit casts if this becomes a problem.

Happy hacking!

Robert Kern

robert.kern@gmail.com

The following section was generated from `docs/tutorials/g2-quick-start.ipynb` .....

CHAPTER

FIVE

QUICK START (G2)

This notebook gives a terse introduction to using the `clifford` module, using a two-dimensional geometric algebra as the context.

5.1 Setup

First, import clifford and instantiate a two-dimensional algebra (G2),

[1]:
```
from numpy import e,pi
import clifford as cf

layout, blades = cf.Cl(2) # creates a 2-dimensional clifford algebra
```

Inspect blades.

[2]:
```
blades
```

[2]:
```
{'': 1, 'e1': (1^e1), 'e2': (1^e2), 'e12': (1^e12)}
```

Assign blades to variables

[3]:
```
e1 = blades['e1']
e2 = blades['e2']
e12 = blades['e12']
```

5.2 Basics

[4]:
```
e1*e2 # geometric product
```

[4]:
```
(1^e12)
```

[5]:
```
e1|e2 # inner product
```

[5]:
```
0
```

[6]:
```
e1^e2 # outer product
```

[6]:
```
(1^e12)
```

5.3 Reflection

[7]:
```
a = e1+e2     # the vector
n = e1        # the reflector
-n*a*n.inv() # reflect `a` in hyperplane normal to `n`
```

[7]:
```
-(1.0^e1) + (1.0^e2)
```

5.4 Rotation

[8]:
```
from numpy  import pi

R = e**(pi/4*e12) # enacts rotation by pi/2
R
```

[8]:
```
0.70711 + (0.70711^e12)
```

[9]:
```
R*e1*~R    # rotate e1 by pi/2 in the e12-plane
```

[9]:
```
-(1.0^e2)
```

..... doc/tutorials/g2-quick-start.ipynb ends here.

The following section was generated from docs/tutorials/g3-algebra-of-space.ipynb .....

CHAPTER

SIX

THE ALGEBRA OF SPACE (G3)

In this notebook, we give a more detailed look at how to use `clifford`, using the algebra of three dimensional space as a context.

6.1 Setup

First, we import clifford as `cf`, and instantiate a three dimensional geometric algebra using `Cl()` (docs).

[1]:
```
import clifford as cf

layout, blades = cf.Cl(3)  # creates a 3-dimensional clifford algebra
```

Given a three dimensional GA with the orthonormal basis,

$$e_i \cdot e_j = \delta_{ij}$$

The basis consists of scalars, three vectors, three bivectors, and a trivector.

$$\{ \underbrace{\alpha,}_{\text{scalar}} \quad \underbrace{e_1, \quad e_2, \quad e_3,}_{\text{vectors}} \quad \underbrace{e_{12}, \quad e_{23}, \quad e_{13},}_{\text{bivectors}} \quad \underbrace{e_{123}}_{\text{trivector}} \}$$

`Cl()` creates the algebra and returns a `layout` and `blades`. The `layout` holds information and functions related this instance of `G3`, and the `blades` is a dictionary which contains the basis blades, indexed by their string representations,

[2]:
```
blades
```

[2]:
```
{'': 1,
 'e1': (1^e1),
 'e2': (1^e2),
 'e3': (1^e3),
 'e12': (1^e12),
 'e13': (1^e13),
 'e23': (1^e23),
 'e123': (1^e123)}
```

You may wish to explicitly assign the blades to variables like so,

[3]:
```
e1 = blades['e1']
e2 = blades['e2']
# etc ...
```

Or, if you're lazy and just working in an interactive session you can use `locals()` to update your namespace with all of the blades at once.

```
[4]: locals().update(blades)
```

Now, all the blades have been defined in the local namespace

```
[5]: e3, e123
[5]: ((1^e3), (1^e123))
```

6.2 Basics

6.2.1 Products

The basic products are available

```
[6]: e1*e2  # geometric product
[6]: (1^e12)
```

```
[7]: e1|e2  # inner product
[7]: 0
```

```
[8]: e1^e2  # outer product
[8]: (1^e12)
```

```
[9]: e1^e2^e3  # even more outer products
[9]: (1^e123)
```

6.2.2 Defects in Precedence

Python's operator precedence makes the outer product evaluate after addition. This requires the use of parentheses when using outer products. For example

```
[10]: e1^e2 + e2^e3  # fail, evaluates as
[10]: (2^e123)
```

```
[11]: (e1^e2) + (e2^e3)  # correct
[11]: (1^e12) + (1^e23)
```

Also the inner product of a scalar and a Multivector is 0,

```
[12]: 4|e1
[12]: 0
```

So for scalars, use the outer product or geometric product instead

```
[13]: 4*e1
[13]: (4^e1)
```

6.2.3 Multivectors

Multivectors can be defined in terms of the basis blades. For example you can construct a rotor as a sum of a scalar and bivector, like so

[14]:
```
from scipy import cos, sin

theta = pi/4
R = cos(theta) - sin(theta)*e23
R
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-7dddb582d1e8> in <module>
      1 from scipy import cos, sin
      2
----> 3 theta = pi/4
      4 R = cos(theta) - sin(theta)*e23
      5 R

NameError: name 'pi' is not defined
```

You can also mix grades without any reason

[15]:
```
A = 1 + 2*e1 + 3*e12 + 4*e123
A
```

[15]:
```
1 + (2^e1) + (3^e12) + (4^e123)
```

6.2.4 Reversion

The reversion operator is accomplished with the tilde ~ in front of the Multivector on which it acts

[16]:
```
~A
```

[16]:
```
1 + (2^e1) - (3^e12) - (4^e123)
```

6.2.5 Grade Projection

Taking a projection onto a specific grade n of a Multivector is usually written

$$\langle A \rangle_n$$

can be done by using soft brackets, like so

[17]:
```
A(0)  # get grade-0 elements of R
```

[17]:
```
1
```

[18]:
```
A(1)  # get grade-1 elements of R
```

[18]:
```
(2^e1)
```

[19]:
```
A(2)  # you get it
```

[19]:
```
(3^e12)
```

6.2.6 Magnitude

Using the reversion and grade projection operators, we can define the magnitude of A

$$|A|^2 = \langle A\tilde{A}\rangle$$

[20]:
```
(A*~A)(0)
```

[20]:
```
30
```

This is done in the `abs()` operator

[21]:
```
abs(A)**2
```

[21]:
```
30.0
```

6.2.7 Inverse

The inverse of a Multivector is defined as $A^{-1}A = 1$

[22]:
```
A.inv()*A
```

[22]:
```
1.0
```

[23]:
```
A.inv()
```

[23]:
```
0.13415 + (0.12195^e1) - (0.14634^e3) + (0.18293^e12) + (0.09756^e23) - (0.29268^e123)
```

6.2.8 Dual

The dual of a multivector A can be defined as

$$AI^{-1}$$

Where, I is the pseudoscalar for the GA. In G_3, the dual of a vector is a bivector,

[24]:
```
a = 1*e1 + 2*e2 + 3*e3
a.dual()
```

[24]:
```
-(3.0^e12) + (2.0^e13) - (1.0^e23)
```

6.2.9 Pretty, Ugly, and Display Precision

You can toggle pretty printing with with `pretty()` or `ugly()`. `ugly` returns an eval-able string.

[25]:
```
cf.ugly()
A.inv()
```

[25]:
```
MultiVector(Layout([1, 1, 1],
                   ids=BasisVectorIds.ordered_integers(3),
                   order=BasisBladeOrder.shortlex(3),
                   names=['', 'e1', 'e2', 'e3', 'e12', 'e13', 'e23', 'e123']),
            [0.13414634146341464, 0.12195121951219513, -0.0, -0.14634146341463417, 0.
↪18292682926829273, -2.0816681711721682e-17, 0.0975609756097561, -0.
↪29268292682926833])
```

You can also change the displayed precision

[26]:
```
cf.pretty(precision=2)

A.inv()
```

[26]:
```
0.13 + (0.12^e1) - (0.15^e3) + (0.18^e12) + (0.1^e23) - (0.29^e123)
```

This does not effect the internal precision used for computations.

6.3 Applications

6.3.1 Reflections

Reflecting a vector c about a normalized vector n is pretty simple,

$$c \to ncn$$

[27]:
```
c = e1+e2+e3    # a vector
n = e1          # the reflector
n*c*n          # reflect `a` in hyperplane normal to `n`
```

[27]:
```
(1^e1) - (1^e2) - (1^e3)
```

Because we have the `inv()` available, we can equally well reflect in un-normalized vectors using,

$$a \to nan^{-1}$$

[28]:
```
a = e1+e2+e3    # the vector
n = 3*e1          # the reflector
n*a*n.inv()
```

[28]:
```
(1.0^e1) - (1.0^e2) - (1.0^e3)
```

Reflections can also be made with respect to the a 'hyperplane normal to the vector n', in this case the formula is negated

$$c \to -ncn^{-1}$$

6.3.2 Rotations

A vector can be rotated using the formula

$$a \to RaR$$

Where R is a rotor. A rotor can be defined by multiple reflections,

$$R = mn$$

or by a plane and an angle,

$$R = e^{-\frac{\theta}{2}\hat{B}}$$

For example

[29]:
```
import math

R = math.e**(-math.pi/4*e12) # enacts rotation by pi/2
R
```

[29]:
```
0.71 - (0.71^e12)
```

[30]:
```
R*e1*~R    # rotate e1 by pi/2 in the e12-plane
```

[30]:
```
(1.0^e2)
```

6.3.3 Some Ways to use Functions

Maybe we want to define a function which can return rotor of some angle θ in the e_{12}-plane,

$$R_{12} = e^{-\frac{\theta}{2}e_{12}}$$

[31]:
```
R12 = lambda theta: e**(-theta/2*e12)
R12(pi/2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-31-517400d2f6b9> in <module>
      1 R12 = lambda theta: e**(-theta/2*e12)
----> 2 R12(pi/2)

NameError: name 'pi' is not defined
```

And use it like this

[32]:
```
a = e1+e2+e3
R = R12(math.pi/2)
R*a*~R
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-32-62fda315a2cc> in <module>
      1 a = e1+e2+e3
```

(continues on next page)

(continued from previous page)

```
----> 2 R = R12(math.pi/2)
      3 R*a*~R

<ipython-input-31-517400d2f6b9> in <lambda>(theta)
----> 1 R12 = lambda theta: e**(-theta/2*e12)
      2 R12(pi/2)

NameError: name 'e' is not defined
```

You might as well make the angle argument a bivector, so that you can control the plane of rotation as well as the angle

$$R_B = e^{-\frac{B}{2}}$$

[33]:
```
R_B = lambda B: math.e**(-B/2)
```

Then you could do

[34]:
```
R12 = R_B(math.pi/4*e12)
R23 = R_B(math.pi/5*e23)
```

or

[35]:
```
R_B(math.pi/6*(e23+e12))  # rotor enacting a pi/6-rotation in the e23+e12-plane
```

[35]:
```
0.93 - (0.26^e12) - (0.26^e23)
```

Maybe you want to define a function which returns a *function* that enacts a specified rotation,

$$f(B) \to \underline{R_B}(a) = R_B a \tilde{R_B}$$

This just saves you having to write out the sandwich product, which is nice if you are cascading a bunch of rotors, like so

$$\underline{R_C}(\underline{R_B}(\underline{R_A}(a)))$$

[36]:
```
def R_factory(B):
    def apply_rotation(a):
        R = math.e**(-B/2)
        return R*a*~R
    return apply_rotation

R = R_factory(pi/6*(e23+e12))  # this returns a function
R(a)  # which acts on a vector
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-36-402a96f6039a> in <module>
      5     return apply_rotation
      6
----> 7 R = R_factory(pi/6*(e23+e12))  # this returns a function
      8 R(a)  # which acts on a vector

NameError: name 'pi' is not defined
```

Then you can do things like

[37]:
```
R12 = R_factory(math.pi/3*e12)
R23 = R_factory(math.pi/3*e23)
R13 = R_factory(math.pi/3*e13)

R12(R23(R13(a)))
```

[37]:
```
(0.41^e1) - (0.66^e2) + (1.55^e3)
```

To make cascading a sequence of rotations as concise as possible, we could define a function which takes a list of bivectors $A, B, C, ..$, and enacts the sequence of rotations which they represent on a some vector x.

$$f(A, B, C, x) = \underline{R_A}(\underline{R_B}(\underline{R_C}(x)))$$

[38]:
```
from functools import reduce

# a sequence of rotations
def R_seq(*args):
    *Bs, x = args
    R_lst = [math.e**(-B/2) for B in Bs]  # create list of Rotors from list of␣
↪Bivectors
    R = reduce(cf.gp, R_lst)          # apply the geometric product to list of Rotors
    return R*x*~R

# rotation sequence by  pi/2-in-e12 THEN pi/2-in-e23
R_seq(pi/2*e23, pi/2*e12, e1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-38-21fc9c28e3a1> in <module>
      9
     10 # rotation sequence by  pi/2-in-e12 THEN pi/2-in-e23
---> 11 R_seq(pi/2*e23, pi/2*e12, e1)

NameError: name 'pi' is not defined
```

6.4 Changing Basis Names

If you want to use different names for your basis as opposed to e's with numbers, supply the `Cl()` with a list of `names`. For example for a two dimensional GA,

[39]:
```
layout,blades = cf.Cl(2, names=['','x','y','i'])

blades
```

[39]:
```
{'': 1, 'x': (1^x), 'y': (1^y), 'i': (1^i)}
```

[40]:
```
locals().update(blades)
```

[41]:
```
1*x + 2*y
```

[41]:
```
(1^x) + (2^y)
```

[42]:
```
1 + 4*i
```

[42]:
```
1 + (4^i)
```

..... doc/tutorials/g3-algebra-of-space.ipynb ends here.

The following section was generated from docs/tutorials/euler-angles.ipynb .....

CHAPTER

SEVEN

ROTATIONS IN SPACE: EULER ANGLES, MATRICES, AND QUATERNIONS

This notebook demonstrates how to use `clifford` to implement rotations in three dimensions using euler angles, rotation matices and quaternions. All of these forms are derived from the more general rotor form, which is provided by GA. Conversion from the rotor form to a matrix representation is shown, and takes about three lines of code. We start with euler angles.

7.1 Euler Angles with Rotors

A common way to parameterize rotations in three dimensions is through Euler Angles.

> Any orientation can be achieved by composing three elemental rotations. The elemental rotations can either occur about the axes of the fixed coordinate system (*extrinsic rotations*) or about the axes of a rotating coordinate system, which is initially aligned with the fixed one, and modifies its orientation after each elemental rotation (*intrinsic rotations*). — wikipedia

The animation below shows an intrinsic rotation model as each elemental rotation is applied.Label the left, right, and vertical blue-axes as e_1, e_2, and e_3, respectively. The series of rotations can be described:

1. rotate about e_3-axis
2. rotate about the rotated e_1-axis, call it $e_1^{'}$
3. rotate about the twice rotated axis of e_3-axis, call it $e_3^{''}$

So the elemental rotations are about $e_3, e_1^{'}, e_3^{''}$-axes, respectively.

(taken from wikipedia)

Following Sec. 2.7.5 from "Geometric Algebra for Physicists", we first rotate an angle ϕ about the e_3-axis, which is equivalent to rotating in the e_{12}-plane. This is done with the rotor

$$R_\phi = e^{-\frac{\phi}{2} e_{12}}$$

Next we rotate about the rotated e_1-axis, which we label $e_1^{'}$. To find where this is, we can rotate the axis,

$$e_1^{'} = R_\phi e_1 \tilde{R_\phi}$$

The plane corresponding to this axis is found by taking the dual of $e_1^{'}$

$$I R_\phi e_1 \tilde{R_\phi} = R_\phi e_{23} \tilde{R_\phi}$$

Where we have made use of the fact that the pseudo-scalar commutes in G3. Using this result, the second rotation by angle θ about the $e_1^{'}$-axis is then ,

$$R_\theta = e^{\frac{\theta}{2} R_\phi e_{23} \tilde{R_\phi}}$$

However, noting that

$$e^{R_\phi e_{23} \tilde{R}_\phi} = R_\phi e^{e_{23}} \tilde{R}_\phi$$

Allows us to write the second rotation by angle θ about the e_1'-axis as

$$R_\theta = R_\phi e^{\frac{\theta}{2} e_{23}} \tilde{R}_\phi$$

So, the combination of the first two elemental rotations equals,

$$\begin{aligned} R_\theta R_\phi &= R_\phi e^{\frac{\theta}{2} e_{23}} \tilde{R}_\phi R_\phi \\ &= e^{-\frac{\phi}{2} e_{12}} e^{-\frac{\theta}{2} e_{23}} \end{aligned}$$

This pattern can be extended to the third elemental rotation of angle ψ in the twice-rotated e_1-axis, creating the total rotor

$$R = e^{-\frac{\phi}{2} e_{12}} e^{-\frac{\theta}{2} e_{23}} e^{-\frac{\psi}{2} e_{12}}$$

7.2 Implementation of Euler Angles

First, we initialize the algebra and assign the variables

[1]:
```
from numpy import e,pi
from clifford import Cl

layout, blades = Cl(3)    # create a 3-dimensional clifford algebra

locals().update(blades) # lazy way to put entire basis in the namespace
```

Next we define a function to produce a rotor given euler angles

[2]:
```
def R_euler(phi, theta,psi):
    Rphi = e**(-phi/2.*e12)
    Rtheta = e**(-theta/2.*e23)
    Rpsi = e**(-psi/2.*e12)

    return Rphi*Rtheta*Rpsi
```

For example, using this to create a rotation similar to that shown in the animation above,

[3]:
```
R = R_euler(pi/4, pi/4, pi/4)
R
```

[3]:
```
0.65328 - (0.65328^e12) - (0.38268^e23)
```

7.3 Convert to Quaternions

A Rotor in 3D space **is** a *unit quaternion*, and so we have essentially created a function that converts Euler angles to quaternions. All you need to do is interpret the bivectors as i, j, and k's. See *Interfacing Other Mathematical Systems*, for more on quaternions.

7.4 Convert to Rotation Matrix

The matrix representation for a rotation can defined as the result of rotating an ortho-normal frame. Rotating an ortho-normal frame can be done easily,

[4]:
```
A = [e1,e2,e3]           # initial ortho-normal frame
B = [R*a*~R for a in A] # resultant frame after rotation

B
```

[4]:
```
[(0.14645^e1) + (0.85355^e2) + (0.5^e3),
 -(0.85355^e1) - (0.14645^e2) + (0.5^e3),
 (0.5^e1) - (0.5^e2) + (0.70711^e3)]
```

The components of this frame **are** the rotation matrix, so we just enter the frame components into a matrix.

[5]:
```
from numpy import array

M = [float(b|a) for b in B for a in A] # you need float() due to bug in clifford
M = array(M).reshape(3,3)

M
```

[5]:
```
array([[ 0.14644661,  0.85355339,  0.5       ],
       [-0.85355339, -0.14644661,  0.5       ],
       [ 0.5       , -0.5       ,  0.70710678]])
```

Thats a rotation matrix.

7.5 Convert a Rotation Matrix to a Rotor

In 3 Dimenions, there is a simple formula which can be used to directly transform a rotations matrix into a rotor. For arbitrary dimensions you have to use a different algorithm (see `clifford.tools.orthoMat2Versor()` (docs)). Anyway, in 3 dimensions there is a closed form solution, as described in Sec. 4.3.3 of "Geometric Algebra for Physicists". Given a rotor R which transforms an orthonormal frame $A = a_k$ into $B = b_k$ as such,

$$b_k = R a_k \tilde{R}$$

R is given by

$$R = \frac{1 + a_k b_k}{|1 + a_k b_k|}$$

So, if you want to convert from a rotation matrix into a rotor, start by converting the matrix `M` into a frame B.(You could do this with loop if you want.)

[6]:
```
B = [M[0,0]*e1 + M[1,0]*e2 + M[2,0]*e3,
     M[0,1]*e1 + M[1,1]*e2 + M[2,1]*e3,
     M[0,2]*e1 + M[1,2]*e2 + M[2,2]*e3]
B
```

[6]:
```
[(0.14645^e1) - (0.85355^e2) + (0.5^e3),
 (0.85355^e1) - (0.14645^e2) - (0.5^e3),
 (0.5^e1) + (0.5^e2) + (0.70711^e3)]
```

Then implement the formula

[7]:
```
A = [e1,e2,e3]
R = 1+sum([A[k]*B[k] for k in range(3)])
R = R/abs(R)

R
```

[7]:
```
0.65328 - (0.65328^e12) - (0.38268^e23)
```

blam.

doc/tutorials/euler-angles.ipynb ends here.

The following section was generated from docs/tutorials/space-time-algebra.ipynb

CHAPTER

EIGHT

SPACE TIME ALGEBRA

8.1 Intro

This notebook demonstrates how to use `clifford` to work with Space Time Algebra. The Pauli algebra of space $\mathbb{P}$, and Dirac algebra of space-time $\mathbb{D}$, are related using the *spacetime split*. The split is implemented by using a `BladeMap` (docs), which maps a subset of blades in $\mathbb{D}$ to the blades in $\mathbb{P}$. This *split* allows a spacetime bivector F to be broken up into relative electric and magnetic fields in space. Lorentz transformations are implemented as rotations in $\mathbb{D}$, and the effects on the relative fields are computed with the split.

8.2 Setup

First we import `clifford`, instantiate the two algebras, and populate the namespace with the blades of each algebra. The elements of $\mathbb{D}$ are prefixed with d, while the elements of $\mathbb{P}$ are prefixed with p. Although unconventional, it is easier to read and to translate into code.

[1]:
```
from clifford import Cl, pretty

pretty(precision=1)

# Dirac Algebra  `D`
D, D_blades = Cl(1,3, firstIdx=0, names='d')

# Pauli Algebra  `P`
P, P_blades = Cl(3, names='p')

# put elements of each in namespace
locals().update(D_blades)
locals().update(P_blades)
```

8.3 The Space Time Split

To two algebras can be related by the spacetime-split. First, we create a `BladeMap` which relates the bivectors in $\mathbb{D}$ to the vectors/bivectors in $\mathbb{P}$. The scalars and pseudo-scalars in each algebra are equated.

[2]:
```
from clifford import BladeMap

bm = BladeMap([(d01,p1),
```

(continues on next page)

(continued from previous page)

```
                (d02,p2),
                (d03,p3),
                (d12,p12),
                (d23,p23),
                (d13,p13),
                (d0123, p123)])
```

8.4 Splitting a space-time vector (an event)

A vector in $\mathbb{D}$, represents a unique place in space and time, i.e. an event. To illustrate the split, create a random event X.

[3]:
```
X = D.randomV()*10
X
```

[3]:
```
(6.8^d0) - (8.7^d1) + (2.7^d2) - (8.5^d3)
```

This can be *split* into time and space components by multiplying with the time-vector d_0,

[4]:
```
X*d0
```

[4]:
```
6.8 + (8.7^d01) - (2.7^d02) + (8.5^d03)
```

and applying the `BladeMap`, which results in a scalar+vector in $\mathbb{P}$

[5]:
```
bm(X*d0)
```

[5]:
```
6.8 + (8.7^p1) - (2.7^p2) + (8.5^p3)
```

The space and time components can be separated by grade projection,

[6]:
```
x = bm(X*d0)
x(0) # the time component
```

[6]:
```
6.8
```

[7]:
```
x(1) # the space component
```

[7]:
```
(8.7^p1) - (2.7^p2) + (8.5^p3)
```

We therefor define a `split()` function, which has a simple condition allowing it to act on a vector or a multivector in $\mathbb{D}$. Splitting a spacetime bivector will be treated in the next section.

[8]:
```
def split(X):
    return bm(X.odd*d0+X.even)
```

[9]:
```
split(X)
```

[9]:
```
6.8 + (8.7^p1) - (2.7^p2) + (8.5^p3)
```

The split can be inverted by applying the `BladeMap` again, and multiplying by d_0

[10]:
```
x = split(X)
bm(x)*d0
```

[10]:
```
(6.8^d0) - (8.7^d1) + (2.7^d2) - (8.5^d3)
```

8.5 Splitting a Bivector

Given a random bivector F in $\mathbb{D}$,

[11]:
```
F = D.randomMV()(2)
F
```

[11]:
```
(0.1^d01) - (1.6^d02) - (1.5^d03) - (0.8^d12) + (1.0^d13) - (0.1^d23)
```

F *splits* into a vector/bivector in $\mathbb{P}$

[12]:
```
split(F)
```

[12]:
```
(0.1^p1) - (1.6^p2) - (1.5^p3) - (0.8^p12) + (1.0^p13) - (0.1^p23)
```

If F is interpreted as the electromagnetic bivector, the Electric and Magnetic fields can be separated by grade

[13]:
```
E = split(F)(1)
iB = split(F)(2)

E
```

[13]:
```
(0.1^p1) - (1.6^p2) - (1.5^p3)
```

[14]:
```
iB
```

[14]:
```
-(0.8^p12) + (1.0^p13) - (0.1^p23)
```

8.6 Lorentz Transformations

Lorentz Transformations are rotations in $\mathbb{D}$, which are implemented with Rotors. A rotor in G4 will, in general, have scalar, bivector, and quadvector components.

[15]:
```
R = D.randomRotor()
R
```

[15]:
```
-0.1 + (0.0^d01) - (0.0^d02) - (0.2^d03) - (0.1^d12) - (1.0^d13) + (0.2^d23) + (0.0^
↪d0123)
```

In this way, the effect of a lorentz transformation on the electric and magnetic fields can be computed by rotating the bivector with $F \to RF\tilde{R}$

[16]:
```
F_ = R*F*~R
F_
```

[16]:
```
-(0.3^d01) - (1.0^d02) + (2.4^d03) + (0.4^d12) + (1.8^d13) - (0.3^d23)
```

Then splitting into E and B fields

[17]:
```
E_ = split(F_)(1)
E_
```

[17]:
```
-(0.3^p1) - (1.0^p2) + (2.4^p3)
```

[18]:
```
iB_ = split(F_)(2)
iB_
```

[18]:
```
(0.4^p12) + (1.8^p13) - (0.3^p23)
```

8.7 Lorentz Invariants

Since lorentz rotations in $\mathbb{D}$, the magnitude of elements of $\mathbb{D}$ are invariants of the lorentz transformation. For example, the magnitude of electromagnetic bivector F is invariant, and it can be related to E and B fields in $\mathbb{P}$ through the split,

[19]:
```
i = p123
E = split(F)(1)
B = -i*split(F)(2)
```

[20]:
```
F**2
```

[20]:
```
3.4 + (5.6^d0123)
```

[21]:
```
split(F**2) == E**2 - B**2 + (2*E|B)*i
```

[21]:
```
True
```

..... doc/tutorials/space-time-algebra.ipynb ends here.

The following section was generated from docs/tutorials/InterfacingOtherMathSystems.ipynb .....

CHAPTER

NINE

INTERFACING OTHER MATHEMATICAL SYSTEMS

Geometric Algebra is known as a *universal algebra* because it subsumes several other mathematical systems. Two algebras commonly used by engineers and scientists are complex numbers and quaternions. These algebras can be subsumed as the even sub-algebras of 2 and 3 dimensional geometric algebras, respectively. This notebook demonstrates how `clifford` can be used to incorporate data created with these systems into geometric algebra.

9.1 Complex Numbers

Given a two dimensional GA with the orthonormal basis,

$$e_i \cdot e_j = \delta_{ij}$$

The geometric algebra consists of scalars, two vectors, and a bivector,

$$\{ \underbrace{\alpha,}_{\text{scalar}} \quad \underbrace{e_1, \quad e_2,}_{\text{vector}} \quad \underbrace{e_{12}}_{\text{bivector}} \}$$

A complex number can be directly associated with a 2D spinor in the e_{12}-plane,

$$\underbrace{\mathbf{z} = \alpha + \beta i}_{\text{complex number}} \quad \Longrightarrow \quad \underbrace{Z = \alpha + \beta e_{12}}_{\text{2D spinor}}$$

The even subalgebra of a two dimensional geometric algebra is isomorphic to the complex numbers. We can setup translating functions which converts a 2D spinor into a complex number and vice-versa. In two dimensions the spinor can be also be mapped into vectors if desired.

Below is an illustration of the three different planes, the later two being contained within the geometric algebra of two dimensions, G_2. Both spinors and vectors in G_2 can be modeled as points on a plane, but they have distinct algebraic properties.

[1]:
```
import clifford as cf
layout, blades = cf.Cl(2) # instantiate a 2D- GA
locals().update(blades)   # put all blades into local namespace


def c2s(z):
    '''convert a complex number to a spinor'''
    return z.real + z.imag*e12

def s2c(S):
    '''convert a spinor to a complex number'''
    S0 = float(S(0))
    S2 = float(-S|e12)
    return S0 + S2*1j
```

Convert a complex number to a spinor

[2]:
```
c2s(1+2j)
```

[2]:
```
1.0 + (2.0^e12)
```

Convert a spinor to a complex number

[3]:
```
s2c(1+2*e12)
```

[3]:
```
(1+2j)
```

Make sure we get what we started with when we make a round trip

[4]:
```
s2c(c2s(1+2j)) == 1+2j
```

[4]:
```
True
```

The spinor is then mapped to a vector by choosing a reference direction. This may be done by left multiplying with e_1 .

$$Z \implies e_1 Z = e_1\alpha + \beta e_1 e_{12} = \underbrace{\alpha e_1 + \beta e_2}_{\text{vector}}$$

[5]:
```
s = 1+2*e12
e1*s
```

[5]:
```
(1^e1) + (2^e2)
```

Geometrically, this is interpreted as having the spinor rotate a specific vector, in this case e_1. Building off of the previously defined functions

[6]:
```
def c2v(c):
    '''convert a complex number to a vector'''
    return e1*c2s(c)

def v2c(v):
    '''convert a vector to a complex number'''
    return s2c(e1*v)
```

[7]:
```
c2v(1+2j)
```

[7]:
```
(1.0^e1) + (2.0^e2)
```

[8]:
```
v2c(1*e1+2*e2)
```

[8]:
```
(1+2j)
```

Depending on your applications, you may wish to have the bivector be an argument to the `c2s` and `s2c` functions. This allows you to map input data given in the form of complex number onto the planes of your choice. For example, in three dimensional space there are three bivector-planes; e_{12}, e_{23} and e_{13}, so there are many bivectors which could be interpreted as the unit imaginary.

Complex numbers mapped in this way can be used to enact rotations within the specified planes.

[9]:
```
import clifford as cf
layout, blades = cf.Cl(3)
locals().update(blades)
```

(continues on next page)

(continued from previous page)

```
def c2s(z,B):
    '''convert a complex number to a spinor'''
    return z.real + z.imag*B

def s2c(S,B):
    '''convert a spinor to a complex number'''
    S0 = float(S(0))
    S2 = float(-S|B)
    return S0 + S2*1j
```

[10]:
```
c2s(1+2j, e23)
```

[10]:
```
1.0 + (2.0^e23)
```

[11]:
```
c2s(3+4j, e13)
```

[11]:
```
3.0 + (4.0^e13)
```

This brings us to the subject of quaternions, which are used to handle rotations in three dimensions much like complex numbers do in two dimensions. With geometric algebra, they are just spinors acting in a different geometry.

9.2 Quaternions

Note:

There is support for quaternions in numpy through the package quaternion.

For some reason people think quaternions (wiki page) are mystical or something. They are just spinors in a three dimensional geometric algebra.

In either case, we can pass the `names` parameters to `Cl()` to explicitly label the bivectors `i`, `j`, and `k`.

[12]:
```
import clifford as cf

# the vector/bivector order is reversed because Hamilton defined quaternions using a
# left-handed frame. wtf.
names = ['', 'z', 'y', 'x', 'k', 'j', 'i', 'I']

layout, blades = cf.Cl(3, names=names)
locals().update(blades)
```

This leads to the commutations relations familiar to quaternion users

[13]:
```
for m in [i, j, k]:
    for n in [i, j, k]:
        print ('{}*{}={}'.format(m, n, m*n))
```

```
(1^i)*(1^i)=-1
(1^i)*(1^j)=(1^k)
(1^i)*(1^k)=-(1^j)
(1^j)*(1^i)=-(1^k)
(1^j)*(1^j)=-1
(1^j)*(1^k)=(1^i)
```

(continues on next page)

(continued from previous page)

```
(1^k)*(1^i)=(1^j)
(1^k)*(1^j)=-(1^i)
(1^k)*(1^k)=-1
```

Quaternion data could be stored in a variety of ways. Assuming you have the scalar components for the quaternion, all you will need to do is setup a map each component to the correct bivector.

[14]:
```
def q2S(*args):
    '''convert tuple of quaternion coefficients to a spinor'''
    q = args
    return q[0] + q[1]*i + q[2]*j + q[3]*k
```

Then all the quaternion computations can be done using GA

[15]:
```
q1 = q2S(1,2,3,4)
q1
```

[15]:
```
1 + (4^k) + (3^j) + (2^i)
```

This prints i, j and k in reverse order but whatever,

[16]:
```
# 'scalar' part
q1(0)
```

[16]:
```
1
```

[17]:
```
# 'vector' part (more like bivector part!)
q1(2)
```

[17]:
```
(4^k) + (3^j) + (2^i)
```

quaternion conjugation is implemented with reversion

[18]:
```
~q1
```

[18]:
```
1 - (4^k) - (3^j) - (2^i)
```

The norm

[19]:
```
abs(q1)
```

[19]:
```
5.477225575051661
```

Taking the `dual()` of the "vector" part actually returns a vector,

[20]:
```
q1(2).dual()
```

[20]:
```
(2.0^z) - (3.0^y) + (4.0^x)
```

[21]:
```
q1 = q2S(1, 2, 3, 4)
q2 = q2S(5, 6, 7, 8)

# quaternion product
q1*q2
```

[21]:
```
-60 + (24^k) + (30^j) + (12^i)
```

If you want to keep using a left-handed frame and names like i, j and k to label the planes in 3D space, ok. If you think it makes more sense to use the consistent and transparent approach provided by GA, we think you have good taste. If we make this switch, the basis and `q2S()` functions will be changed to

[22]:
```
import clifford as cf
layout, blades = cf.Cl(3)
locals().update(blades)

blades
```

[22]:
```
{'': 1,
 'e1': (1^e1),
 'e2': (1^e2),
 'e3': (1^e3),
 'e12': (1^e12),
 'e13': (1^e13),
 'e23': (1^e23),
 'e123': (1^e123)}
```

[23]:
```
def q2S(*args):
    '''
    convert tuple of quaternion coefficients to a spinor'''
    q = args
    return q[0] + q[1]*e13 +q[2]*e23 + q[3]*e12

q1 = q2S(1,2,3,4)
q1
```

[23]:
```
1 + (4^e12) + (2^e13) + (3^e23)
```

..... doc/tutorials/InterfacingOtherMathSystems.ipynb ends here.

The following section was generated from docs/tutorials/PerformanceCliffordTutorial.ipynb .....

CHAPTER

TEN

WRITING HIGH(ISH) PERFORMANCE CODE WITH CLIFFORD AND NUMBA VIA NUMPY

This document describes how to take algorithms developed in the clifford package with notation that is close to the maths and convert it into numerically efficient and fast running code. To do this we will expose the underlying representation of multivector as a numpy array of canonical basis vector coefficients and operate directly on these arrays in a manner that is conducive to JIT compilation with numba.

10.1 First import the Clifford library as well as numpy and numba

[1]:
```
import clifford as cf
import numpy as np
import numba
```

10.2 Choose a specific space

For this document we will use 3d euclidean space embedded in the conformal framework giving a Cl(4,1) algebra. We will also rename some of the variables to match the notation that used by Lasenby et al. in "A Covariant Approach to Geometry using Geometric Algebra"

[2]:
```
from clifford import g3c
# 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
```

10.3 Performance of mathematically idiomatic Clifford algorithms

By default the Clifford library sacrifices performance for syntactic convenience.

Consider a function that applies a rotor to a multivector:

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

We will define a rotor that takes one line to another:

[4]:
```
line_one = (up(0)^up(e1)^ninf).normal()
line_two = (up(0)^up(e2)^ninf).normal()
R = 1 + line_two*line_one
```

Check that this works

[5]:
```
print(line_two)
print(apply_rotor(R,line_one).normal())
```

```
(1.0^e245)
(1.0^e245)
```

We would like to improve the speed of our algorithm, first we will profile it and see where it spends its time

[6]:
```
#%%prun -s cumtime
#for i in range(1000000):
#    apply_rotor(R,line_one)
```

An example profile output from running this notebook on the author's laptop is as follows:

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000   66.290   66.290 {built-in method builtins.exec}
      1    0.757    0.757   66.290   66.290 <string>:2(<module>)
1000000    3.818    0.000   65.534    0.000 <ipython-input-13-70a01003bf51>:1(apply_
↪rotor)
2000000    9.269    0.000   55.641    0.000 __init__.py:751(__mul__)
2000000    3.167    0.000   29.900    0.000 __init__.py:717(_checkOther)
2000000    1.371    0.000   19.906    0.000 __init__.py:420(__ne__)
2000000    6.000    0.000   18.535    0.000 numeric.py:2565(array_equal)
2000000   10.505    0.000   10.505    0.000 __init__.py:260(mv_mult)
```

We can see that the function spends almost all of its time in `__mul__` and within `__mul__` it spends most of its time in `_checkOther`. In fact it only spends a small fraction of its time in `mv_mult` which does the numerical multivector multiplication. To write more performant code we need to strip away the high level abstractions and deal with the underlying representations of the blade component data.

10.4 Canonical blade coefficient representation in Clifford

In Clifford a multivector is internally represented as a numpy array of the coefficients of the canonical basis vectors, they are arranged in order of grade. So for our 4,1 algebra the first element is the scalar part, the next 5 are the vector coefficients, the next 10 are the bivectors, the next 10 the triectors, the next 5 the quadvectors and the final value is the pseudoscalar coefficient.

[7]:
```
(5.0*e1 - e2 + e12 + e135 + np.pi*e1234).value
```

[7]:
```
array([ 0.        ,  5.        , -1.        ,  0.        ,  0.        ,
        0.        ,  1.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        1.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  3.14159265,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ])
```

10.5 Exploiting blade representation to write a fast function

We can rewrite our rotor application function using the functions that the layout exposes for operations on the numpy arrays themselves.

[8]:
```
def apply_rotor_faster(R,mv):
    return layout.MultiVector(layout.gmt_func(R.value,layout.gmt_func(mv.value,layout.
↪adjoint_func(R.value))) )
```

[9]:
```
#%%prun -s cumtime
#for i in range(1000000):
#    apply_rotor_faster(R,line_one)
```

This gives a much faster function

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000   19.567   19.567 {built-in method builtins.exec}
      1    0.631    0.631   19.567   19.567 <string>:2(<module>)
1000000    7.373    0.000   18.936    0.000 <ipython-input-35-6a5344d83bdb>:1(apply_
↪rotor_faster)
2000000    9.125    0.000    9.125    0.000 __init__.py:260(mv_mult)
1000000    1.021    0.000    1.619    0.000 __init__.py:677(__init__)
1000000    0.819    0.000    0.819    0.000 __init__.py:244(adjoint_func)
```

We have successfully skipped past the higher level checks on the multivectors while maintaining exactly the same function signature. It is important to check that we still have the correct answer:

[10]:
```
print(line_two)
print(apply_rotor_faster(R,line_one).normal())
```

```
(1.0^e245)
(1.0^e245)
```

The performance improvements gained by rewriting our function are significant but it comes at the cost of readability.

By loading the layouts gmt_func and adjoint_func into the global namespace before the function is defined and separating the value operations from the multivector wrapper we can make our code more concise.

[11]:
```
gmt_func = layout.gmt_func
adjoint_func = layout.adjoint_func

def apply_rotor_val(R_val,mv_val):
    return gmt_func(R_val,gmt_func(mv_val,adjoint_func(R_val)))

def apply_rotor_wrapped(R,mv):
    return cf.MultiVector(layout,apply_rotor_val(R.value,mv.value))
```

[12]:
```
#%%prun -s cumtime
#for i in range(1000000):
#    apply_rotor_wrapped(R,line_one)
```

The time cost is essentially the same, there is probably some minor overhead from the function call itself

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   19.621   19.621 {built-in method builtins.exec}
     1    0.557    0.557   19.621   19.621 <string>:2(<module>)
1000000    1.421    0.000   19.064    0.000 <ipython-input-38-a1e0b5c53cdc>:7(apply_
↪rotor_wrapped)
1000000    6.079    0.000   16.033    0.000 <ipython-input-38-a1e0b5c53cdc>:4(apply_
↪rotor_val)
2000000    9.154    0.000    9.154    0.000 __init__.py:260(mv_mult)
1000000    1.017    0.000    1.610    0.000 __init__.py:677(__init__)
1000000    0.800    0.000    0.800    0.000 __init__.py:244(adjoint_func)
```

[13]:
```
print(line_two)
print(apply_rotor_wrapped(R,line_one).normal())
```

```
(1.0^e245)
(1.0^e245)
```

The additional advantage of splitting the function like this is that the numba JIT compiler can reason about the memory layout of numpy arrays in no python mode as long as no pure python objects are operated upon within the function. This means we can JIT our function that operates on the value directly.

[14]:
```
@numba.njit
def apply_rotor_val_numba(R_val,mv_val):
    return gmt_func(R_val,gmt_func(mv_val,adjoint_func(R_val)))

def apply_rotor_wrapped_numba(R,mv):
    return cf.MultiVector(layout,apply_rotor_val_numba(R.value,mv.value))
```

[15]:
```
#%%prun -s cumtime
#for i in range(1000000):
#    apply_rotor_wrapped_numba(R,line_one)
```

This gives a small improvement in performance but more importantly it allows us to write larger functions that also use the jitted `apply_rotor_val_numba` and are themselves jitted.

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   16.033   16.033 {built-in method builtins.exec}
     1    0.605    0.605   16.033   16.033 <string>:2(<module>)
 1000000    2.585    0.000   15.428    0.000 <ipython-input-42-1142126d93ca>:5(apply_
↪rotor_wrapped_numba)
 1000000    8.606    0.000    8.606    0.000 <ipython-input-42-1142126d93ca>:1(apply_
↪rotor_val_numba)
```

(continues on next page)

(continued from previous page)

```
      1    0.000    0.000    2.716    2.716 dispatcher.py:294(_compile_for_args)
    7/1    0.000    0.000    2.716    2.716 dispatcher.py:554(compile)
```

10.6 Composing larger functions

By chaining together functions that operate on the value arrays of multivectors it is easy to construct fast and readable code

[16]:
```
I5_val = I5.value
omt_func = layout.omt_func

def dual_mv(mv):
    return -I5*mv

def meet_unwrapped(mv_a,mv_b):
    return -dual_mv(dual_mv(mv_a)^dual_mv(mv_b))

@numba.njit
def dual_val(mv_val):
    return -gmt_func(I5_val,mv_val)

@numba.njit
def meet_val(mv_a_val,mv_b_val):
    return -dual_val( omt_func( dual_val(mv_a_val) , dual_val(mv_b_val)) )

def meet_wrapped(mv_a,mv_b):
    return cf.layout.MultiVector(meet_val(mv_a.value, mv_b.value))

sphere = (up(0)^up(e1)^up(e2)^up(e3)).normal()
print(sphere.meet(line_one).normal().normal())
print(meet_unwrapped(sphere,line_one).normal())
print(meet_wrapped(line_one,sphere).normal())
```

```
(1.0^e14) - (1.0^e15) - (1.0^e45)
(1.0^e14) - (1.0^e15) - (1.0^e45)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-059d06919085> in <module>
     22 print(sphere.meet(line_one).normal().normal())
     23 print(meet_unwrapped(sphere,line_one).normal())
---> 24 print(meet_wrapped(line_one,sphere).normal())

<ipython-input-16-059d06919085> in meet_wrapped(mv_a, mv_b)
     17
     18 def meet_wrapped(mv_a,mv_b):
---> 19     return cf.layout.MultiVector(meet_val(mv_a.value, mv_b.value))
     20
     21 sphere = (up(0)^up(e1)^up(e2)^up(e3)).normal()

AttributeError: module 'clifford' has no attribute 'layout'
```

[17]:
```
#%%prun -s cumtime
#for i in range(100000):
#    meet_unwrapped(sphere,line_one)
```

```
ncalls tottime percall cumtime percall filename:lineno(function) 1 0.
000 0.000 13.216 13.216 {built-in method builtins.exec} 1 0.085 0.
085 13.216 13.216 <string>:2(<module>) 100000 0.418 0.000 13.131 0.000
<ipython-input-98-f91457c8741a>:7(meet_unwrapped) 300000 0.681 0.000 9.893
0.000 <ipython-input-98-f91457c8741a>:4(dual_mv) 300000 1.383 0.000 8.127
0.000 __init__.py:751(__mul__) 400000 0.626 0.000 5.762 0.000 __init__.py:
717(_checkOther) 400000 0.270 0.000 3.815 0.000 __init__.py:420(__ne__) 400000
1.106 0.000 3.544 0.000 numeric.py:2565(array_equal) 100000 0.460 0.000 2.439
0.000 __init__.py:783(__xor__) 800000 0.662 0.000 2.053 0.000 __init__.py:
740(_newMV) 400000 1.815 0.000 1.815 0.000 __init__.py:260(mv_mult)
```

[18]:
```
#%%prun -s cumtime
#for i in range(100000):
#    meet_wrapped(sphere,line_one)
```

```
ncalls tottime percall cumtime percall filename:lineno(function) 1
0.000 0.000 1.951 1.951 {built-in method builtins.exec} 1 0.063 0.
063 1.951 1.951 <string>:2(<module>) 100000 0.274 0.000 1.888 0.000
<ipython-input-98-f91457c8741a>:18(meet_wrapped) 100000 1.448 0.000 1.448
0.000 <ipython-input-98-f91457c8741a>:14(meet_val) 100000 0.096 0.000 0.166
0.000 __init__.py:677(__init__)
```

10.7 Algorithms exploiting known sparseness of MultiVector value array

The standard multiplication generator function for two general multivectors is as follows:

[19]:
```
def get_mult_function(mult_table,n_dims):
    '''
    Returns a function that implements the mult_table on two input multivectors
    '''
    non_zero_indices = mult_table.nonzero()
    k_list = non_zero_indices[0]
    l_list = non_zero_indices[1]
    m_list = non_zero_indices[2]
    mult_table_vals = np.array([mult_table[k,l,m] for k,l,m in np.transpose(non_zero_
↪indices)],dtype=int)

    @numba.njit
    def mv_mult(value,other_value):
        output = np.zeros(n_dims)
        for ind,k in enumerate(k_list):
            l = l_list[ind]
            m = m_list[ind]
            output[l] += value[k]*mult_table_vals[ind]*other_value[m]
        return output
    return mv_mult
```

There are however instances in which we might be able to use the known sparseness of the input data value representation to speed up the operations. For example, in Cl(4,1) rotors only contain even grade blades and we can therefore remove all the operations accessing odd grade objects.

[20]:
```
def get_grade_from_index(index_in):
    if index_in == 0:
```

(continues on next page)

(continued from previous page)

```
        return 0
    elif index_in < 6:
        return 1
    elif index_in < 16:
        return 2
    elif index_in < 26:
        return 3
    elif index_in < 31:
        return 4
    elif index_in == 31:
        return 5
    else:
        raise ValueError('Index is out of multivector bounds')

def get_sparse_mult_function(mult_table,n_dims,grades_a,grades_b):
    '''
    Returns a function that implements the mult_table on two input multivectors
    '''
    non_zero_indices = mult_table.nonzero()
    k_list = non_zero_indices[0]
    l_list = non_zero_indices[1]
    m_list = non_zero_indices[2]
    mult_table_vals = np.array([mult_table[k,l,m] for k,l,m in np.transpose(non_zero_
↪indices)],dtype=int)

    # Now filter out the sparseness
    filter_mask = np.zeros(len(k_list), dtype=bool)
    for i in range(len(filter_mask)):
        if get_grade_from_index(k_list[i]) in grades_a:
            if get_grade_from_index(m_list[i]) in grades_b:
                filter_mask[i] = 1

    k_list = k_list[filter_mask]
    l_list = l_list[filter_mask]
    m_list = m_list[filter_mask]
    mult_table_vals = mult_table_vals[filter_mask]

    @numba.njit
    def mv_mult(value,other_value):
        output = np.zeros(n_dims)
        for ind,k in enumerate(k_list):
            l = l_list[ind]
            m = m_list[ind]
            output[l] += value[k]*mult_table_vals[ind]*other_value[m]
        return output
    return mv_mult
```

[21]:
```
left_rotor_mult = get_sparse_mult_function(layout.gmt,layout.gaDims,[0,2,4],[0,1,2,3,
↪4,5])
right_rotor_mult = get_sparse_mult_function(layout.gmt,layout.gaDims,[0,1,2,3,4,5],[0,
↪2,4])

@numba.njit
def sparse_apply_rotor_val(R_val,mv_val):
    return left_rotor_mult(R_val,right_rotor_mult(mv_val,adjoint_func(R_val)))
```

(continues on next page)

(continued from previous page)

```
def sparse_apply_rotor(R,mv):
    return cf.MultiVector(layout,sparse_apply_rotor_val(R.value,mv.value))
```

[22]:
```
#%%prun -s cumtime
#for i in range(1000000):
#    sparse_apply_rotor(R,line_one)
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    9.490    9.490 {built-in method builtins.exec}
      1    0.624    0.624    9.489    9.489 <string>:2(<module>)
1000000    2.684    0.000    8.865    0.000 <ipython-input-146-f75aae3ce595>:
↪8(sparse_apply_rotor)
1000000    4.651    0.000    4.651    0.000 <ipython-input-146-f75aae3ce595>:
↪4(sparse_apply_rotor_val)
1000000    0.934    0.000    1.530    0.000 __init__.py:677(__init__)
1000000    0.596    0.000    0.596    0.000 {built-in method numpy.core.multiarray.
↪array}
```

[23]:
```
print(line_two)
print(sparse_apply_rotor(R,line_one).normal())
```

```
(1.0^e245)
(1.0^e245)
```

We can do the same with the meet operation that we defined earlier if we know what grade objects we are meeting

[24]:
```
left_pseudo_mult = get_sparse_mult_function(layout.gmt,layout.gaDims,[5],[0,1,2,3,4,
↪5])
sparse_omt_2_1 = get_sparse_mult_function(layout.omt,layout.gaDims,[2],[1])

@numba.njit
def dual_sparse_val(mv_val):
    return -left_pseudo_mult(I5_val,mv_val)

@numba.njit
def meet_sparse_3_4_val(mv_a_val,mv_b_val):
    return -dual_sparse_val( sparse_omt_2_1( dual_sparse_val(mv_a_val) , dual_sparse_
↪val(mv_b_val)) )

def meet_sparse_3_4(mv_a,mv_b):
    return cf.layout.MultiVector(meet_sparse_3_4_val(mv_a.value, mv_b.value))

print(sphere.meet(line_one).normal().normal())
print(meet_sparse_3_4(line_one,sphere).normal())
```

```
(1.0^e14) - (1.0^e15) - (1.0^e45)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-24-b25f7cce90d8> in <module>
     14
     15 print(sphere.meet(line_one).normal().normal())
---> 16 print(meet_sparse_3_4(line_one,sphere).normal())

<ipython-input-24-b25f7cce90d8> in meet_sparse_3_4(mv_a, mv_b)
     11
     12 def meet_sparse_3_4(mv_a,mv_b):
```

(continues on next page)

(continued from previous page)

```
---> 13     return cf.layout.MultiVector(meet_sparse_3_4_val(mv_a.value, mv_b.value))
     14
     15 print(sphere.meet(line_one).normal().normal())

AttributeError: module 'clifford' has no attribute 'layout'
```

[25]:
```
#%%prun -s cumtime
#for i in range(100000):
#    meet_sparse_3_4(line_one,sphere)
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.725    0.725 {built-in method builtins.exec}
     1    0.058    0.058    0.725    0.725 <string>:2(<module>)
100000    0.252    0.000    0.667    0.000 <ipython-input-156-f346d0563682>:12(meet_
↪sparse_3_4)
100000    0.267    0.000    0.267    0.000 <ipython-input-156-f346d0563682>:8(meet_
↪sparse_3_4_val)
100000    0.088    0.000    0.148    0.000 __init__.py:677(__init__)
```

10.8 Future work on performance

Investigate efficient operations on containers of large numbers of multivectors.

Possibly investigate http://numba.pydata.org/numba-doc/0.13/CUDAJit.html for larger algebras/other areas in which GPU memory latency will not be such a large factor, ie, lots of bulk parallel numerical operations.

[ ]:
```
```

doc/tutorials/PerformanceCliffordTutorial.ipynb ends here.

The following section was generated from docs/tutorials/cga/index.ipynb

CHAPTER

ELEVEN

CONFORMAL GEOMETRIC ALGEBRA

Conformal Geometric Algebra (CGA) is a projective geometry tool which allows conformal transformations to be implemented with rotations. To do this, the original geometric algebra is extended by two dimensions, one of positive signature e_+ and one of negative signature e_-. Thus, if we started with G_p, the conformal algebra is $G_{p+1,1}$.

It is convenient to define a *null* basis given by

$$e_o = \frac{1}{2}(e_- - e_+)$$
$$e_\infty = e_- + e_+$$

A vector in the original space x is *up-projected* into a conformal vector X by

$$X = x + \frac{1}{2}x^2 e_\infty + e_o$$

To map a conformal vector back into a vector from the original space, the vector is first normalized, then rejected from the minkowski plane E_0,

$$X = \frac{X}{X \cdot e_\infty}$$

then

$$x = X \wedge E_0 \, E_0^{-1}$$

To implement this in `clifford` we could create a CGA by instantiating the it directly, like `Cl(3,1)` for example, and then making the definitions and maps described above relating the various subspaces. Or, we you can use the helper function `conformalize()`.

11.1 Using `conformalize()`

The purpose of `conformalize()` is to remove the redundancy associated with creating a conformal geometric algebras. `conformalize()` takes an existing geometric algebra layout and *conformalizes* it by adding two dimensions, as described above. Additionally, this function returns a new layout for the CGA, a dict of blades for the CGA, and dictionary containing the added basis vectors and up/down projection functions.

To demonstrate we will conformalize G_2, producing a CGA of $G_{3,1}$.

[1]:
```
from numpy import pi,e
from clifford import Cl, conformalize

G2, blades_g2 = Cl(2)

blades_g2 # inspect the G2 blades
```

[1]:
```
{'': 1, 'e1': (1^e1), 'e2': (1^e2), 'e12': (1^e12)}
```

Now, conformalize it

[2]:
```
G2c, blades_g2c, stuff = conformalize(G2)

blades_g2c   #inspect the CGA blades
```

[2]:
```
{'': 1,
 'e1': (1^e1),
 'e2': (1^e2),
 'e3': (1^e3),
 'e4': (1^e4),
 'e12': (1^e12),
 'e13': (1^e13),
 'e14': (1^e14),
 'e23': (1^e23),
 'e24': (1^e24),
 'e34': (1^e34),
 'e123': (1^e123),
 'e124': (1^e124),
 'e134': (1^e134),
 'e234': (1^e234),
 'e1234': (1^e1234)}
```

Additionally lets inspect `stuff`

[3]:
```
stuff
```

[3]:
```
{'ep': (1^e3),
 'en': (1^e4),
 'eo': -(0.5^e3) + (0.5^e4),
 'einf': (1^e3) + (1^e4),
 'E0': (1.0^e34),
 'up': <bound method ConformalLayout.up of ConformalLayout([1, 1, 1, -1],␣
↪ids=BasisVectorIds.ordered_integers(4), order=BasisBladeOrder.shortlex(4), names=['
↪', 'e1', 'e2', 'e3', 'e4', 'e12', 'e13', 'e14', 'e23', 'e24', 'e34', 'e123', 'e124',
↪ 'e134', 'e234', 'e1234'])>,
 'down': <bound method ConformalLayout.down of ConformalLayout([1, 1, 1, -1],␣
↪ids=BasisVectorIds.ordered_integers(4), order=BasisBladeOrder.shortlex(4), names=['
↪', 'e1', 'e2', 'e3', 'e4', 'e12', 'e13', 'e14', 'e23', 'e24', 'e34', 'e123', 'e124',
↪ 'e134', 'e234', 'e1234'])>,
 'homo': <bound method ConformalLayout.homo of ConformalLayout([1, 1, 1, -1],␣
↪ids=BasisVectorIds.ordered_integers(4), order=BasisBladeOrder.shortlex(4), names=['
↪', 'e1', 'e2', 'e3', 'e4', 'e12', 'e13', 'e14', 'e23', 'e24', 'e34', 'e123', 'e124',
↪ 'e134', 'e234', 'e1234'])>,
 'I_base': (1.0^e12)}
```

It contains the following:

- `ep` - positive basis vector added
- `en` - negative basis vector added
- `eo` - zero vector of null basis (=.5*(en-ep))
- `einf` - infinity vector of null basis (=en+ep)
- `E0` - minkowski bivector (=einf^eo)
- `up()` - function to up-project a vector from GA to CGA

- `down()` - function to down-project a vector from CGA to GA
- `homo()` - function to homogenize a CGA vector

We can put the `blades` and the `stuff` into the local namespace,

[4]:
```
locals().update(blades_g2c)
locals().update(stuff)
```

Now we can use the `up()` and `down()` functions to go in and out of CGA

[5]:
```
x = e1+e2
X = up(x)
X
```

[5]:
```
(1.0^e1) + (1.0^e2) + (0.5^e3) + (1.5^e4)
```

[6]:
```
down(X)
```

[6]:
```
(1.0^e1) + (1.0^e2)
```

11.2 Operations

Conformal transformations in G_n are achieved through versors in the conformal space $G_{n+1,1}$. These versors can be categorized by their relation to the added minkowski plane, E_0. There are three categories,

- versor purely in E_0
- versor partly in E_0
- versor out of E_0

A three dimensional projection for conformal space with the relevant subspaces labeled is shown below.

11.2.1 Versors purely in E_0

First we generate some vectors in G2, which we can operate on

[7]:
```
a= 1*e1 + 2*e2
b= 3*e1 + 4*e2
```

Inversions

$$e_+ X e_+$$

Inversion is a reflection in e_+, this swaps e_o and e_∞, as can be seen from the model above.

[8]:
```
assert(down(ep*up(a)*ep)  == a.inv())
```

Involutions

$$E_0 X E_0$$

[9]:
```
assert(down(E0*up(a)*E0) == -a)
```

Dilations

$$D_\alpha = e^{-\frac{\ln \alpha}{2} E_0}$$

$$D_\alpha X \tilde{D_\alpha}$$

[10]:
```
from scipy import rand,log

D = lambda alpha: e**((-log(alpha)/2.)*(E0))
alpha = rand()
assert(down( D(alpha)*up(a)*~D(alpha)) == (alpha*a))
```

```
<ipython-input-10-0e2e8e068b72>:4: DeprecationWarning: scipy.rand is deprecated and
↪will be removed in SciPy 2.0.0, use numpy.random.rand instead
  alpha = rand()
<ipython-input-10-0e2e8e068b72>:3: DeprecationWarning: scipy.log is deprecated and
↪will be removed in SciPy 2.0.0, use numpy.lib.scimath.log instead
  D = lambda alpha: e**((-log(alpha)/2.)*(E0))
```

11.2.2 Versors partly in E_0

Translations

$$V = e^{\frac{1}{2} e_\infty a} = 1 + e_\infty a$$

[11]:
```
T = lambda x: e**(1/2.*(einf*x))
assert(down( T(a)*up(b)*~T(a)) == b+a)
```

Transversions

A transversion is an inversion, followed by a translation, followed by a inversion. The versor is

$$V = e_+ T_a e_+$$

which is recognised as the translation bivector reflected in the e_+ vector. From the diagram, it is seen that this is equivalent to the bivector in $x \wedge e_o$,

$$e_+(1 + e_\infty a)e_+$$

$$e_+^2 + e_+ e_\infty a e_+$$

$$2 + 2e_o a$$

the factor of 2 may be dropped, because the conformal vectors are null

[12]:
```
V = ep * T(a) * ep
assert ( V == 1+(eo*a))

K = lambda x: 1+(eo*a)

B= up(b)
assert( down(K(a)*B*~K(a)) == 1/(a+1/b) )
```

11.2.3 Versors Out of E_0

Versors that are out of E_0 are made up of the versors within the original space. These include reflections and rotations, and their conformal representation is identical to their form in G^n, except the minus sign is dropped for reflections,

Reflections

$$-mam^{-1} \to MA\tilde{M}$$

[13]:
```
m = 5*e1 + 6*e2
n = 7*e1 + 8*e2


assert(down(m*up(a)*m) == -m*a*m.inv())
```

Rotations

$$mnanm = Ra\tilde{R} \to RA\tilde{R}$$

[14]:
```
R = lambda theta: e**((-.5*theta)*(e12))
theta = pi/2
assert(down( R(theta)*up(a)*~R(theta)) == R(theta)*a*~R(theta))
```

11.2.4 Combinations of Operations

simple example

As a simple example consider the combination operations of translation,scaling, and inversion.

$$b = -2a + e_0 \quad \to \quad B = (T_{e_0}E_0D_2)A(D_2\tilde{E_0}T_{e_0})$$

[15]:
```
A = up(a)
V = T(e1)*E0*D(2)
B = V*A*~V
assert(down(B) == (-2*a)+e1 )
```

```
<ipython-input-10-0e2e8e068b72>:3: DeprecationWarning: scipy.log is deprecated and
↪will be removed in SciPy 2.0.0, use numpy.lib.scimath.log instead
  D = lambda alpha: e**((-log(alpha)/2.)*(E0))
```

Transversion

A transversion may be built from a inversion, translation, and inversion.

$$c = (a^{-1} + b)^{-1}$$

In conformal GA, this is accomplished by

$$C = VA\tilde{V}$$

$$V = e_+ T_b e_+$$

[16]:
```
A = up(a)
V = ep*T(b)*ep
C = V*A*~V
assert(down(C) ==1/(1/a +b))
```

Rotation about a point

Rotation about a point a can be achieved by translating the origin to a, then rotating, then translating back. Just like the transversion can be thought of as translating the involution operator, rotation about a point can also be thought of as translating the Rotor itself. Covariance.

11.3 More examples

The following section was generated from docs/tutorials/cga/visualization-tools.ipynb .....

11.3.1 Visualization tools

In this example we will look at some external tools that can be used with `clifford` to help visualize geometric objects.

The two tools available are:

- `pyganja` (github)
- `mpl_toolkits.clifford` (github)

Both of these can be installed with `pip install` followed by the package name above.

G2C

Let's start by creating some objects in 2d Conformal Geometric Algebra to visualize:

[1]:
```
from clifford.g2c import *
```

[2]:
```
point = up(2*e1+e2)
line = up(3*e1 + 2*e2) ^ up(3*e1 - 2*e2) ^ einf
circle = up(e1) ^ up(-e1 + 2*e2) ^ up(-e1 - 2*e2)
```

We'll create copies of the point and line reflected in the circle, using $X = C\hat{X}\tilde{C}$, where $\hat{X}$ is the grade involution.

[3]:
```
point_refl = circle * point.gradeInvol() * ~circle
line_refl = circle * line.gradeInvol() * ~circle
```

pyganja

pyganja is a python interface to the `ganja.js` (github) library. To use it, typically we need to import two names from the library:

[4]:
```
from pyganja import GanjaScene, draw
import pyganja; pyganja.__version__
```

```
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/pyganja/__init__.py:2: UserWarning: Failed to import cef_gui, cef␣
↪functions will be unavailable
  from .script_api import *
```

[4]:
```
'0.0.12'
```

`GanjaScene` lets us build scenes out of geometric objects, with attached labels and RGB colors:

[5]:
```
sc = GanjaScene()
sc.add_object(point, color=(255, 0, 0), label='point')
sc.add_object(line, color=(0, 255, 0), label='line')
sc.add_object(circle, color=(0, 0, 255), label='circle')
```

[6]:
```
sc_refl = GanjaScene()
sc_refl.add_object(point_refl, color=(128, 0, 0), label='point_refl')
sc_refl.add_object(line_refl, color=(0, 128, 0), label='line_refl')
```

Once we've built our scene, we can `draw` it, specifying a `scale` (which here we use to zoom out), and the signature of our algebra (which defaults to conformal 3D):

[7]:
```
draw(sc, sig=layout.sig, scale=0.5)
```

```
<IPython.core.display.Javascript object>
```

A cool feature of `GanjaScene` is the ability to use + to draw both scenes together:

[8]:
```
draw(sc + sc_refl, sig=layout.sig, scale=0.5)
```

```
<IPython.core.display.Javascript object>
```

mpl_toolkits.clifford

While `ganja.js` produces great diagrams, it's hard to combine them with other plotting tools. `mpl_toolkits.clifford` works within `matplotlib`.

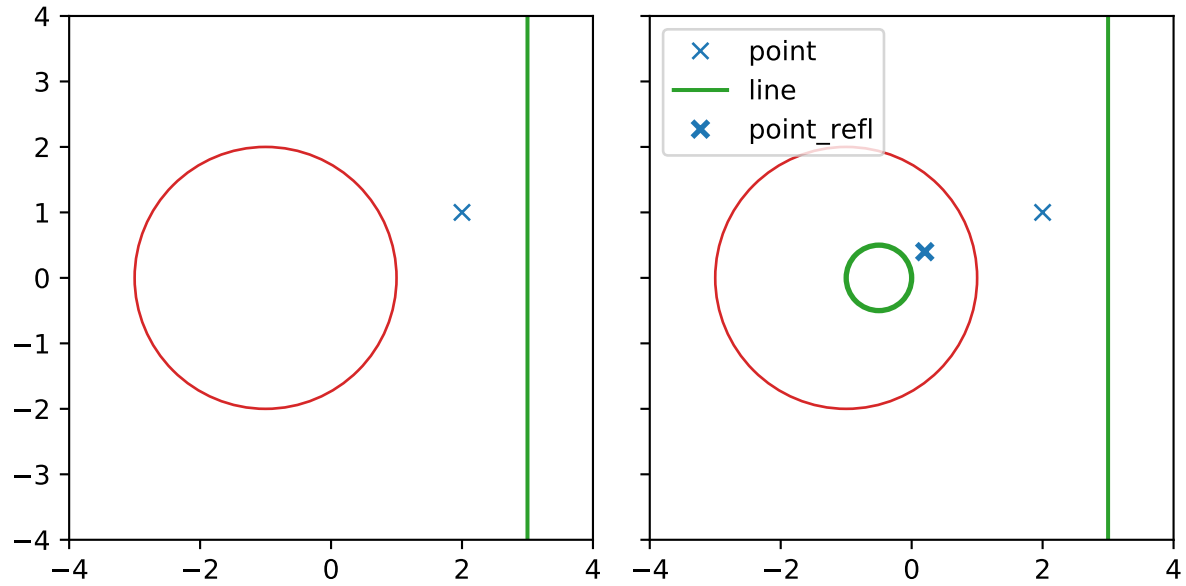
[9]:
```
from matplotlib import pyplot as plt
plt.ioff()  # we'll ask for plotting when we want it

# if you're editing this locally, you'll get an interactive UI if you uncomment the following
#
#    %matplotlib notebook

from mpl_toolkits.clifford import plot
import mpl_toolkits.clifford; mpl_toolkits.clifford.__version__
```

[9]:
```
'0.0.3'
```

Assembling the plot is a lot more work, but we also get much more control:

[10]:
```
# standard matplotlib stuff - construct empty plots side-by-side, and set the scaling
fig, (ax_before, ax_both) = plt.subplots(1, 2, sharex=True, sharey=True)
ax_before.set(xlim=[-4, 4], ylim=[-4, 4], aspect='equal')
ax_both.set(xlim=[-4, 4], ylim=[-4, 4], aspect='equal')

# plot the objects before reflection on both plots
for ax in (ax_before, ax_both):
    plot(ax, [point], color='tab:blue', label='point', marker='x', linestyle=' ')
    plot(ax, [line], color='tab:green', label='line')
    plot(ax, [circle], color='tab:red', label='circle')

# plot the objects after reflection, with thicker lines
plot(ax_both, [point_refl], color='tab:blue', label='point_refl',  marker='x', linestyle=' ', markeredgewidth=2)
plot(ax_both, [line_refl], color='tab:green', label='line_refl', linewidth=2)

fig.tight_layout()
ax_both.legend()

# show the figure
fig
```

[10]:



G3C

Let's repeat the above, but with 3D Conformal Geometric Algebra. Note that if you're viewing these docs in a jupyter notebook, the lines below will replace all your 2d variables with 3d ones

[11]:
```
from clifford.g3c import *
```

[12]:
```
point = up(2*e1+e2)
line = up(3*e1 + 2*e2) ^ up(3*e1 - 2*e2) ^ einf
circle = up(e1) ^ up(-e1 + 1.6*e2 + 1.2*e3) ^ up(-e1 - 1.6*e2 - 1.2*e3)
sphere = up(3*e1) ^ up(e1) ^ up(2*e1 + e2) ^ up(2*e1 + e3)
```

[13]:
```
# note that due to floating point rounding, we need to truncate back to a single␣
↪grade here, with ``(grade)``
point_refl = homo((circle * point.gradeInvol() * ~circle)(1))
line_refl = (circle * line.gradeInvol() * ~circle)(3)
sphere_refl = (circle * sphere.gradeInvol() * ~circle)(4)
```

pyganja

Once again, we can create a pair of scenes exactly as before

[14]:
```
sc = GanjaScene()
sc.add_object(point, color=(255, 0, 0), label='point')
sc.add_object(line, color=(0, 255, 0), label='line')
sc.add_object(circle, color=(0, 0, 255), label='circle')
sc.add_object(sphere, color=(0, 255, 255), label='sphere')
```

[15]:
```
sc_refl = GanjaScene()
sc_refl.add_object(point_refl, color=(128, 0, 0), label='point_refl')
```

(continues on next page)

(continued from previous page)

```
sc_refl.add_object(line_refl.normal(), color=(0, 128, 0), label='line_refl')
sc_refl.add_object(sphere_refl.normal(), color=(0, 128, 128), label='sphere_refl')
```

But this time, when we draw them we don't need to pass `sig`. Better yet, we can rotate the 3D world around using left click, pan with right click, and zoom with the scroll wheel.

[16]:
```
draw(sc + sc_refl, scale=0.5)
```

```
<IPython.core.display.Javascript object>
```

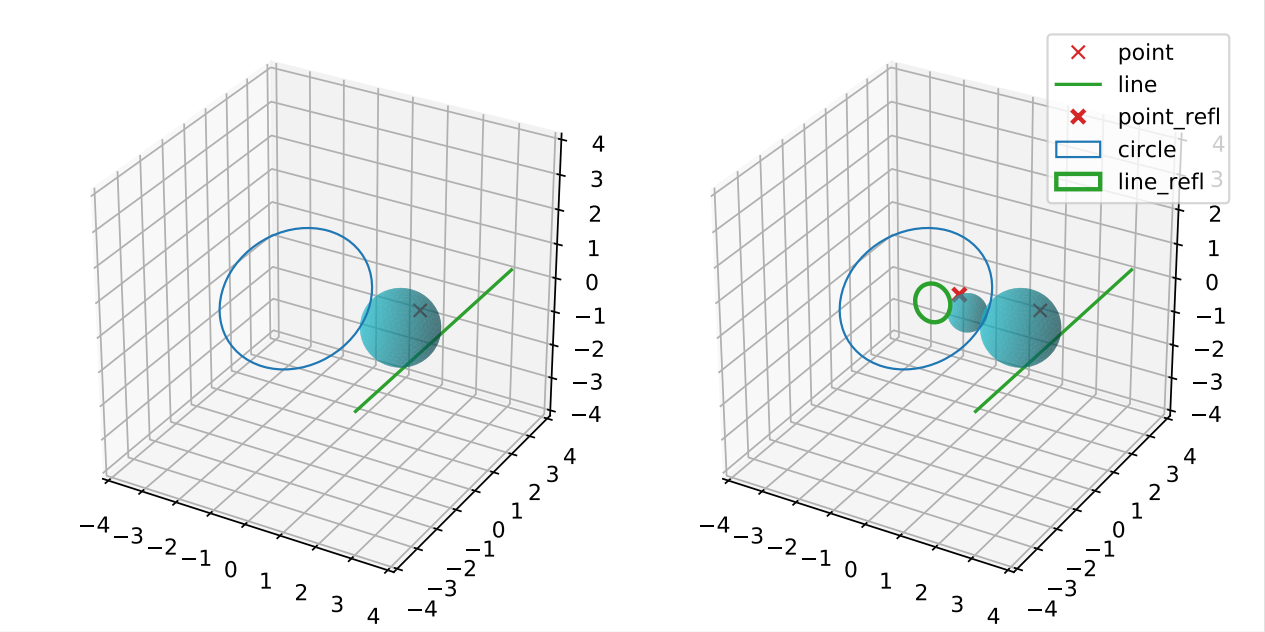
Some more example of using `pyganja` to visualize 3D CGA can be found in the *interpolation* and *clustering* notebooks.

mpl_toolkits.clifford

The 3D approach for `matplotlib` is much the same. Note that due to poor handling of rounding errors in `clifford.tools.classify`, a call to `.normal()` is needed. Along with explicit grade selection, this is a useful trick to try and get something to render which otherwise would not.

[17]:
```
# standard matplotlib stuff - construct empty plots side-by-side, and set the scaling
fig, (ax_before, ax_both) = plt.subplots(1, 2, subplot_kw=dict(projection='3d'),␣
↪figsize=(8, 4))
ax_before.set(xlim=[-4, 4], ylim=[-4, 4], zlim=[-4, 4])
ax_both.set(xlim=[-4, 4], ylim=[-4, 4], zlim=[-4, 4])

# plot the objects before reflection on both plots
for ax in (ax_before, ax_both):
    plot(ax, [point], color='tab:red', label='point', marker='x', linestyle=' ')
    plot(ax, [line], color='tab:green', label='line')
    plot(ax, [circle], color='tab:blue', label='circle')
    plot(ax, [sphere], color='tab:cyan')  # labels do not work for spheres: pygae/mpl_
↪toolkits.clifford#5

# plot the objects after reflection
plot(ax_both, [point_refl], color='tab:red', label='point_refl', marker='x',␣
↪linestyle=' ', markeredgewidth=2)
plot(ax_both, [line_refl.normal()], color='tab:green', label='line_refl', linewidth=2)
plot(ax_both, [sphere_refl], color='tab:cyan')

fig.tight_layout()
ax_both.legend()

# show the figure
fig
```

[17]:



Some more example of using `mpl_toolkits.clifford` to visualize 3D CGA can be found in the `examples` folder of the `mpl_toolkits.clifford` repositiory, here.

doc/tutorials/cga/visualization-tools.ipynb ends here.

The following section was generated from docs/tutorials/cga/object-oriented.ipynb

11.3.2 Object Oriented CGA

This is a shelled out demo for a object-oriented approach to CGA with `clifford`. The `CGA` object holds the original layout for an arbitrary geometric algebra , and the conformalized version. It provides up/down projections, as well as easy ways to generate objects and operators.

Quick Use Demo

[1]:
```
from clifford.cga import CGA, Round, Translation
from clifford import Cl

g3,blades = Cl(3)

cga = CGA(g3)  # make cga from existing ga
# or
cga = CGA(3)   # generate cga from dimension of 'base space'

locals().update(cga.blades)         # put ga's blades in local namespace

C = cga.round(e1,e2,e3,-e2)       # generate unit sphere from points
C
```

```
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/pyganja/__init__.py:2: UserWarning: Failed to import cef_gui, cef␣
↪functions will be unavailable
  from .script_api import *
```

[1]:
```
Sphere
```

[2]:
```
## Objects
cga.round()               # from None
cga.round(3)              # from dim of space
cga.round(e1,e2,e3,-e2)   # from points
cga.round(e1,e2,e3)       # from points
cga.round(e1,e2)          # from points
cga.round((e1,3))         # from center, radius
cga.round(cga.round().mv)# from existing multivector

cga.flat()                # from None
cga.flat(2)               # from dim of space
cga.flat(e1,e2)           # from points
cga.flat(cga.flat().mv)   # from existing multivector


## Operations
cga.dilation()           # from from None
cga.dilation(.4)         # from int

cga.translation()        # from None
cga.translation(e1+e2)   # from vector
cga.translation(cga.down(cga.null_vector()))

cga.rotation()           # from None
cga.rotation(e12+e23)    # from bivector

cga.transversion(e1+e2).mv
```

```
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/clifford/_multivector.py:262: RuntimeWarning: divide by zero␣
↪encountered in true_divide
  newValue = self.value / other
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/clifford/_multivector.py:262: RuntimeWarning: invalid value␣
↪encountered in true_divide
  newValue = self.value / other
```

[2]:
```
1.0 + (0.5^e14) - (0.5^e15) + (0.5^e24) - (0.5^e25)
```

[3]:
```
cga.round().inverted()
```

[3]:
```
-(0.47846^e1234) + (0.12837^e1235) + (1.05451^e1245) - (0.12634^e1345) + (0.29067^
↪e2345)
```

[4]:
```
D = cga.dilation(5)
cga.down(D(e1))
```

[4]:
```
(5.0^e1)
```

[5]:
```
C.mv # any CGA object/operator has a multivector
```

[5]:
```
(1.0^e1235)
```

[6]:
```
C.center_down,C.radius # some properties of spheres
```

[6]:
```
(0, 1.0)
```

[7]:
```
T = cga.translation(e1+e2) # make a translation
C_ = T(C)                  # translate the sphere
cga.down(C_.center)        # compute center again
```

[7]:
```
(1.0^e1) + (1.0^e2)
```

[8]:
```
cga.round()       #  no args == random sphere
cga.translation() #             random translation
```

[8]:
```
Translation
```

[9]:
```
if 1 in map(int, [1,2]):
    print(3)
```

```
3
```

Objects

Vectors

[10]:
```
a = cga.base_vector()  # random vector with components in base space only
a
```

[10]:
```
(0.51296^e1) + (0.95581^e2) + (0.65308^e3)
```

[11]:
```
cga.up(a)
```

[11]:
```
(0.51296^e1) + (0.95581^e2) + (0.65308^e3) + (0.30161^e4) + (1.30161^e5)
```

[12]:
```
cga.null_vector()  # create null vector directly
```

[12]:
```
(1.02882^e1) + (1.81476^e2) - (0.41951^e3) + (1.7639^e4) + (2.7639^e5)
```

Sphere (point pair, circles)

[13]:
```
C = cga.round(e1, e2, -e1, e3) # generates sphere from points
C = cga.round(e1, e2, -e1)     # generates circle from points
C = cga.round(e1, e2)          # generates point-pair from points
#or
C2 = cga.round(2)            # random 2-sphere  (sphere)
C1 = cga.round(1)            # random 1-sphere, (circle)
C0 = cga.round(0)            # random 0-sphere, (point pair)


C1.mv                        # access the multivector
```

[13]:
```
(0.25644^e123) + (0.78834^e124) + (1.09508^e125) + (0.51988^e134) + (0.75719^e135) +␣
↪(0.10771^e145) + (0.4448^e234) + (0.57059^e235) - (0.14535^e245) - (0.15662^e345)
```

[14]:
```
C = cga.round(e1, e2, -e1, e3)
C.center,C.radius        # spheres have properties
```

[14]:
```
(-(1.0^e4) + (1.0^e5), 1.0)
```

[15]:
```
cga.down(C.center) == C.center_down
```

[15]:
```
True
```

[16]:
```
C_ = cga.round().from_center_radius(C.center,C.radius)
C_.center,C_.radius
```

[16]:
```
(-(2.0^e4) + (2.0^e5), 0.9999999999999999)
```

Operators

[17]:
```
T = cga.translation(e1) # generate translation
T.mv
```

[17]:
```
1.0 - (0.5^e14) - (0.5^e15)
```

[18]:
```
C = cga.round(e1, e2, -e1)
T.mv*C.mv*~T.mv          # translate a sphere
```

[18]:
```
-(0.5^e124) + (0.5^e125) - (1.0^e245)
```

[19]:
```
T(C)                # shorthand call, same as above. returns type of arg
```

[19]:
```
Circle
```

[20]:
```
T(C).center
```

[20]:
```
(2.0^e1) + (2.0^e5)
```

[ ]:

[ ]:

..... doc/tutorials/cga/object-oriented.ipynb ends here.

The following section was generated from docs/tutorials/cga/interpolation.ipynb .....

11.3.3 Example 1 Interpolating Conformal Objects

In this example we will look at a few of the tools provided by the clifford package for (4,1) conformal geometric algebra (CGA) and see how we can use them in a practical setting to interpolate geometric primitives.

The first step in using the package for CGA is to generate and import the algebra:

[1]:
```
from clifford.g3c import *
blades
```

[1]:
```
{'': 1,
 'e1': (1^e1),
 'e2': (1^e2),
 'e3': (1^e3),
 'e4': (1^e4),
 'e5': (1^e5),
 'e12': (1^e12),
 'e13': (1^e13),
 'e14': (1^e14),
```

(continues on next page)

(continued from previous page)

```
 'e15': (1^e15),
 'e23': (1^e23),
 'e24': (1^e24),
 'e25': (1^e25),
 'e34': (1^e34),
 'e35': (1^e35),
 'e45': (1^e45),
 'e123': (1^e123),
 'e124': (1^e124),
 'e125': (1^e125),
 'e134': (1^e134),
 'e135': (1^e135),
 'e145': (1^e145),
 'e234': (1^e234),
 'e235': (1^e235),
 'e245': (1^e245),
 'e345': (1^e345),
 'e1234': (1^e1234),
 'e1235': (1^e1235),
 'e1245': (1^e1245),
 'e1345': (1^e1345),
 'e2345': (1^e2345),
 'e12345': (1^e12345)}
```

This creates an algebra with the required signature and imports the basis blades into the current workspace. We can check our metric by squaring our grade 1 multivectors.

[2]:
```
print('e1*e1 ', e1*e1)
print('e2*e2 ', e2*e2)
print('e3*e3 ', e3*e3)
print('e4*e4 ', e4*e4)
print('e5*e5 ', e5*e5)
```

```
e1*e1  1
e2*e2  1
e3*e3  1
e4*e4  1
e5*e5  -1
```

As expected this gives us 4 basis vectors that square to 1.0 and one that squares to -1.0, therefore confirming our metric is (4,1).

The up() function implements the mapping of vectors from standard 3D space to conformal space. We can use this to construct conformal objects to play around with.

For example a line at the origin:

[3]:
```
line_a = ( up(0)^up(e1+e2)^einf ).normal()
print(line_a)
```

```
(0.70711^e145) + (0.70711^e245)
```

The tools submodule of the clifford package contains a wide array of algorithms and tools that can be useful for manipulating objects in CGA. We will use these tools to generate rotors that rotate and translate objects:

[4]:
```
from clifford.tools.g3 import *
from clifford.tools.g3c import *
from numpy import pi
```

(continues on next page)

(continued from previous page)

```
rotation_radians = pi/4
euc_vector_in_plane_m = e1
euc_vector_in_plane_n = e3

euc_translation = -5.2*e1 + 3*e2 - pi*e3

rotor_rotation = generate_rotation_rotor(rotation_radians, euc_vector_in_plane_m, euc_
↪vector_in_plane_n)
rotor_translation = generate_translation_rotor(euc_translation)
print(rotor_rotation)
print(rotor_translation)

combined_rotor = (rotor_translation*rotor_rotation).normal()

line_b = (combined_rotor*line_a*~combined_rotor).normal()
print(line_b)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/pyganja/__init__.py:2: UserWarning: Failed to import cef_gui, cef␣
↪functions will be unavailable
  from .script_api import *
```

```
0.92388 - (0.38268^e13)
1.0 + (2.6^e14) + (2.6^e15) - (1.5^e24) - (1.5^e25) + (1.5708^e34) + (1.5708^e35)
-(5.17696^e124) - (5.17696^e125) - (1.0292^e134) - (1.0292^e135) + (0.5^e145) + (3.
↪72144^e234) + (3.72144^e235) + (0.70711^e245) + (0.5^e345)
```

In the above snippet of code we have generated rotors for translation and rotation, then combined these, then applied the combined rotor to the original line that we made.

Visualizations

The `clifford` package can be used alongside `pyganja` to render CGA objects which can be rotated interactively in a jupyter notebook:

[5]:
```
from pyganja import GanjaScene, draw
sc = GanjaScene()
sc.add_object(line_a,color=0xFF0000, label='a')
sc.add_object(line_b,color=0x00FF00, label='b')
draw(sc, scale=0.1)
```

```
<IPython.core.display.Javascript object>
```

We can also interpolate the objects using the tools in clifford and can visualise the result

[6]:
```
def interpolate_objects_linearly(L1, L2, n_steps=10, color_1=np.array([255,0,0]),␣
↪color_2=np.array([0,255,0])):
    alpha_list = np.linspace(0, 1, num=n_steps)
    intermediary_list = []
    sc = GanjaScene()
    for alpha in alpha_list:
        intermediate_color = (alpha*color_1 + (1-alpha)*color_2).astype(np.uint8)
        intermediate_object = interp_objects_root(L1, L2, alpha)
        intermediary_list.append(intermediate_object)
        color_string = int(
            (intermediate_color[0] << 16) | (intermediate_color[1] << 8) |␣
↪intermediate_color[2]
```

(continues on next page)

(continued from previous page)

```
        )
        sc.add_object(intermediate_object, color_string)
    return intermediary_list, sc
```

[7]:
```
intermediary_list, finished_scene = interpolate_objects_linearly(line_a, line_b)
draw(finished_scene, scale=0.1)
```

```
<IPython.core.display.Javascript object>
```

We can do the same for all the other geometric primitives as well

Circles

[8]:
```
circle_a = (up(0)^up(e1)^up(e2)).normal()
circle_b = (combined_rotor*circle_a*~combined_rotor).normal()
intermediary_list, finished_scene = interpolate_objects_linearly(circle_a, circle_b)
draw(finished_scene, scale=0.1)
```

```
<IPython.core.display.Javascript object>
```

Point pairs

[9]:
```
point_pair_a = (up(e3)^up(e1+e2)).normal()
point_pair_b = (combined_rotor*point_pair_a*~combined_rotor).normal()
intermediary_list, finished_scene = interpolate_objects_linearly(point_pair_a, point_
↪pair_b)
draw(finished_scene, scale=0.1)
```

```
<IPython.core.display.Javascript object>
```

Planes

[10]:
```
plane_a = (up(0)^up(e1)^up(e2)^einf).normal()
plane_b = (combined_rotor*plane_a*~combined_rotor).normal()
intermediary_list, finished_scene = interpolate_objects_linearly(plane_a, plane_b)
draw(finished_scene)
```

```
<IPython.core.display.Javascript object>
```

Spheres

[11]:
```
sphere_a = (up(0)^up(e1)^up(e2)^up(e3)).normal()
sphere_b = (combined_rotor*sphere_a*~combined_rotor).normal()
intermediary_list, finished_scene = interpolate_objects_linearly(sphere_a, sphere_b)
draw(finished_scene, scale=0.1)
```

```
<IPython.core.display.Javascript object>
```

doc/tutorials/cga/interpolation.ipynb ends here.

The following section was generated from docs/tutorials/cga/clustering.ipynb

11.3.4 Example 2 Clustering Geometric Objects

In this example we will look at a few of the tools provided by the clifford package for (4,1) conformal geometric algebra (CGA) and see how we can use them in a practical setting to cluster geometric objects via the simple K-means clustering algorithm provided in clifford.tools

As before the first step in using the package for CGA is to generate and import the algebra:

[1]:
```
from clifford.g3c import *
print('e1*e1 ', e1*e1)
print('e2*e2 ', e2*e2)
print('e3*e3 ', e3*e3)
print('e4*e4 ', e4*e4)
print('e5*e5 ', e5*e5)
```

```
e1*e1  1
e2*e2  1
e3*e3  1
e4*e4  1
e5*e5  -1
```

The tools submodule of the clifford package contains a wide array of algorithms and tools that can be useful for manipulating objects in CGA. In this case we will be generating a large number of objects and then segmenting them into clusters.

We first need an algorithm for generating a cluster of objects in space. We will construct this cluster by generating a random object and then repeatedly disturbing this object by some small fixed amount and storing the result:

[2]:
```
from clifford.tools.g3c import *
import numpy as np

def generate_random_object_cluster(n_objects, object_generator, max_cluster_trans=1.0,
↪ max_cluster_rot=np.pi/8):
    """ Creates a cluster of random objects """
    ref_obj = object_generator()
    cluster_objects = []
    for i in range(n_objects):
        r = random_rotation_translation_rotor(maximum_translation=max_cluster_trans,␣
↪maximum_angle=max_cluster_rot)
        new_obj = apply_rotor(ref_obj, r)
        cluster_objects.append(new_obj)
    return cluster_objects
```

```
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/pyganja/__init__.py:2: UserWarning: Failed to import cef_gui, cef␣
↪functions will be unavailable
  from .script_api import *
```

We can use this function to create a cluster and then we can visualise this cluster with pyganja.

[3]:
```
from pyganja import *
clustered_circles = generate_random_object_cluster(10, random_circle)
sc = GanjaScene()
for c in clustered_circles:
    sc.add_object(c, rgb2hex([255,0,0]))
draw(sc, scale=0.05)
```

```
<IPython.core.display.Javascript object>
```

This cluster generation function appears in clifford tools by default and it can be imported as follows:

[4]:
```
from clifford.tools.g3c import generate_random_object_cluster
```

Now that we can generate individual clusters we would like to generate many:

[5]:
```
def generate_n_clusters( object_generator, n_clusters, n_objects_per_cluster ):
    object_clusters = []
    for i in range(n_clusters):
        cluster_objects = generate_random_object_cluster(n_objects_per_cluster,␣
↪object_generator,
                                                         max_cluster_trans=0.5, max_
↪cluster_rot=np.pi / 16)
        object_clusters.append(cluster_objects)
    all_objects = [item for sublist in object_clusters for item in sublist]
    return all_objects, object_clusters
```

Again this function appears by default in clifford tools and we can easily visualise the result:

[6]:
```
from clifford.tools.g3c import generate_n_clusters

all_objects, object_clusters = generate_n_clusters(random_circle, 2, 5)
sc = GanjaScene()
for c in all_objects:
    sc.add_object(c, rgb2hex([255,0,0]))
draw(sc, scale=0.05)
```

```
<IPython.core.display.Javascript object>
```

Given that we can now generate multiple clusters of objects we can test algorithms for segmenting them.

The function run_n_clusters below generates a lot of objects distributed into n clusters and then attempts to segment the objects to recover the clusters.

[7]:
```
from clifford.tools.g3c.object_clustering import n_clusters_objects
import time

def run_n_clusters( object_generator, n_clusters, n_objects_per_cluster, n_
↪shotgunning):
    all_objects, object_clusters = generate_n_clusters( object_generator, n_clusters,␣
↪n_objects_per_cluster )
    [new_labels, centroids, start_labels, start_centroids] = n_clusters_objects(n_
↪clusters, all_objects,
                                                                                 ␣
↪initial_centroids=None,
                                                                                  n_
↪shotgunning=n_shotgunning,
                                                                                 ␣
↪averaging_method='unweighted')
    return all_objects, new_labels, centroids
```

Lets try it!

[8]:
```
def visualise_n_clusters(all_objects, centroids, labels,
                         color_1=np.array([255, 0, 0]),
                         color_2=np.array([0, 255, 0])):
```

(continues on next page)

(continued from previous page)

```
    """
    Utility method for visualising several clusters and their respective centroids
    using pyganja
    """
    alpha_list = np.linspace(0, 1, num=len(centroids))
    sc = GanjaScene()
    for ind, this_obj in enumerate(all_objects):
        alpha = alpha_list[labels[ind]]
        cluster_color = (alpha * color_1 + (1 - alpha) * color_2).astype(np.int)
        sc.add_object(this_obj, rgb2hex(cluster_color))

    for c in centroids:
        sc.add_object(c, Color.BLACK)

    return sc



object_generator = random_circle

n_clusters = 3
n_objects_per_cluster = 10
n_shotgunning = 60
all_objects, labels, centroids = run_n_clusters(object_generator, n_clusters,
                                                n_objects_per_cluster, n_
↪shotgunning)

sc = visualise_n_clusters(all_objects, centroids, labels,
                          color_1=np.array([255, 0, 0]),
                          color_2=np.array([0, 255, 0]))
draw(sc, scale=0.05)
```

```
<IPython.core.display.Javascript object>
```

..... doc/tutorials/cga/clustering.ipynb ends here.

The following section was generated from docs/tutorials/cga/robotic-manipulators.ipynb .....

11.3.5 Application to Robotic Manipulators

This notebook is intended to expand upon the ideas in part of the presentation Robots, Ganja & Screw Theory

Serial manipulator

(slides)

Let's consider a 2-link 3 DOF arm. We'll model the links within the robot with rotors, which transform to the coordinate frame of the end of each link. This is very similar to the approach that would classically be taken with 4×4 matrices.

We're going to define our class piecewise as we go along here. To aid that, we'll write a simple base class to let us do just that. In your own code, there's no need to do this.

[1]:
```
class AddMethodsAsWeGo:
    @classmethod
```

(continues on next page)

(continued from previous page)

```
    def _add_method(cls, m):
        if isinstance(m, property):
            name = (m.fget or m.fset).__name__
        else:
            name = m.__name__
        setattr(cls, name, m)
```

Let's start by defining some names for the links, and a place to store our parameters:

[2]:
```
from enum import Enum

class Links(Enum):
    BASE = 'b'
    SHOULDER = 's'
    UPPER = 'u'
    ELBOW = 'e'
    FOREARM = 'f'
    ENDPOINT = 'n'


class SerialRobot(AddMethodsAsWeGo):
    def __init__(self, rho, l):
        self.l = l
        self.rho = rho
        self._thetas = (0, 0, 0)

    @property
    def thetas(self):
        return self._thetas
```

Forward kinematics

(slides)

As a reminder, we can construct rotation and translation motors as:

$$
\begin{aligned}
T(a) &= \exp\left(\frac{1}{2} n_\infty \wedge a\right) && (11.1)\\
&= 1 + \frac{1}{2} n_\infty \wedge a && (11.2)\\
R(\theta, \hat{B}) &= \exp(-\frac{1}{2}\theta \hat{B}) && (11.3)\\
&= \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\hat{B} && (11.4)
\end{aligned}
$$

Applying these to our geometry, we get

$$
\begin{aligned}
R_{\text{base}\leftarrow\text{shoulder}} &= R(\theta_0, e_1 \wedge e_3) && (11.5)\\
R_{\text{shoulder}\leftarrow\text{upper arm}} &= R(\theta_1, e_1 \wedge e_2) && (11.6)\\
R_{\text{upper arm}\leftarrow\text{elbow}} &= T(l_1 e_1) && (11.7)\\
R_{\text{elbow}\leftarrow\text{forearm}} &= R(\theta_2, e_1 \wedge e_2) && (11.8)\\
R_{\text{forearm}\leftarrow\text{endpoint}} &= T(l_2 e_1) && (11.9)\\
& && (11.10)
\end{aligned}
$$

From which we can get the overall rotor to the frame of the endpoint, and the positions X and Y:

$$R_{\text{base}\leftarrow\text{elbow}} = R_{\text{base}\leftarrow\text{shoulder}} R_{\text{shoulder}\leftarrow\text{upper arm}} R_{\text{upper arm}\leftarrow\text{elbow}} \tag{11.11}$$

$$X = R_{\text{base}\leftarrow\text{elbow}} n_0 \tilde{R}_{\text{base}\leftarrow\text{elbow}} \tag{11.12}$$

$$R_{\text{base}\leftarrow\text{endpoint}} = R_{\text{base}\leftarrow\text{shoulder}} R_{\text{shoulder}\leftarrow\text{upper arm}} R_{\text{upper arm}\leftarrow\text{elbow}} R_{\text{elbow}\leftarrow\text{forearm}} R_{\text{forearm}\leftarrow\text{endpoint}} \tag{11.13}$$

$$Y = R_{\text{base}\leftarrow\text{endpoint}} n_0 \tilde{R}_{\text{base}\leftarrow\text{endpoint}} \tag{11.14}$$

$$\tag{11.15}$$

We can write this as:

[3]:
```
from clifford.g3c import *
from clifford.tools.g3c import generate_translation_rotor, apply_rotor
from clifford.tools.g3 import generate_rotation_rotor

def _update_chain(rotors, a, b, c):
    rotors[a, c] = rotors[a, b] * rotors[b, c]

@SerialRobot._add_method
@SerialRobot.thetas.setter
def thetas(self, value):
    theta0, theta1, theta2 = self._thetas = value
    # shorthands for brevity
    R = generate_rotation_rotor
    T = generate_translation_rotor

    rotors = {}
    rotors[Links.BASE, Links.SHOULDER] = R(theta0, e1, e3)
    rotors[Links.SHOULDER, Links.UPPER] = R(theta1, e1, e2)
    rotors[Links.UPPER, Links.ELBOW] = T(self.rho * e1)
    rotors[Links.ELBOW, Links.FOREARM] = R(theta2, e1, e2)
    rotors[Links.FOREARM, Links.ENDPOINT] = T(-self.l * e1)

    _update_chain(rotors, Links.BASE, Links.SHOULDER, Links.UPPER)
    _update_chain(rotors, Links.BASE, Links.UPPER, Links.ELBOW)
    _update_chain(rotors, Links.BASE, Links.ELBOW, Links.FOREARM)
    _update_chain(rotors, Links.BASE, Links.FOREARM, Links.ENDPOINT)
    self.rotors = rotors

@SerialRobot._add_method
@property
def y_pos(self):
    return apply_rotor(eo, self.rotors[Links.BASE, Links.ENDPOINT])


@SerialRobot._add_method
@property
def x_pos(self):
    return apply_rotor(eo, self.rotors[Links.BASE, Links.ELBOW])
```

```
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/pyganja/__init__.py:2: UserWarning: Failed to import cef_gui, cef␣
↪functions will be unavailable
  from .script_api import *
```

Let's write a renderer so we can check this all works

[4]:
```
from pyganja import GanjaScene

def add_rotor(sc: GanjaScene, r, *, label=None, color=None, scale=0.1):
    """ show how a rotor transforms the axes at the origin """
    y = apply_rotor(eo, r)
    y_frame = [
        apply_rotor(d, r)
        for d in [up(scale*e1), up(scale*e2), up(scale*e3)]
    ]
    sc.add_object(y, label=label, color=color)
    sc.add_facet([y, y_frame[0]], color=(255, 0, 0))
    sc.add_facet([y, y_frame[1]], color=(0, 255, 0))
    sc.add_facet([y, y_frame[2]], color=(0, 0, 255))


@SerialRobot._add_method
def to_scene(self):
    sc = GanjaScene()
    axis_scale = 0.1
    link_scale = 0.05
    arm_color = (192, 192, 192)

    base_obj = (up(0.2*e1)^up(0.2*e3)^up(-0.2*e1)).normal()
    sc.add_object(base_obj, color=0)

    shoulder_axis = [
        apply_rotor(p, self.rotors[Links.BASE, Links.UPPER])
        for p in [up(axis_scale*e3), up(-axis_scale*e3)]
    ]
    sc.add_facet(shoulder_axis, color=(0, 0, 128))
    shoulder_angle = [
        apply_rotor(eo, self.rotors[Links.BASE, Links.SHOULDER]),
        apply_rotor(up(axis_scale*e1), self.rotors[Links.BASE, Links.SHOULDER]),
        apply_rotor(up(axis_scale*e1), self.rotors[Links.BASE, Links.UPPER]),
    ]
    sc.add_facet(shoulder_angle, color=(0, 0, 128))

    upper_arm_points = [
        apply_rotor(up(link_scale*e3), self.rotors[Links.BASE, Links.UPPER]),
        apply_rotor(up(-link_scale*e3), self.rotors[Links.BASE, Links.UPPER]),
        apply_rotor(up(link_scale*e3), self.rotors[Links.BASE, Links.ELBOW]),
        apply_rotor(up(-link_scale*e3), self.rotors[Links.BASE, Links.ELBOW])
    ]
    sc.add_facet(upper_arm_points[:3], color=arm_color)
    sc.add_facet(upper_arm_points[1:], color=arm_color)

    elbow_axis = [
        apply_rotor(p, self.rotors[Links.BASE, Links.ELBOW])
        for p in [up(axis_scale*e3), up(-axis_scale*e3)]
    ]
    sc.add_facet(elbow_axis, color=(0, 0, 128))

    forearm_points = [
        apply_rotor(up(link_scale*e3), self.rotors[Links.BASE, Links.FOREARM]),
        apply_rotor(up(-link_scale*e3), self.rotors[Links.BASE, Links.FOREARM]),
        apply_rotor(up(link_scale*e3), self.rotors[Links.BASE, Links.ENDPOINT]),
        apply_rotor(up(-link_scale*e3), self.rotors[Links.BASE, Links.ENDPOINT])
```

(continues on next page)

(continued from previous page)

```
    ]
    sc.add_facet(forearm_points[:3], color=arm_color)
    sc.add_facet(forearm_points[1:], color=arm_color)

    add_rotor(sc, self.rotors[Links.BASE, Links.ELBOW], label='x', color=(128, 128,␣
↪128))
    add_rotor(sc, self.rotors[Links.BASE, Links.ENDPOINT], label='y', color=(128, 128,
↪ 128))

    return sc
```

We can now instantiate our robot

[5]:
```
serial_robot = SerialRobot(rho=1, l=0.5)
```

Choose a trajectory

[6]:
```
import math
theta_traj = [
    (math.pi/6 + i*math.pi/12, math.pi/3 - math.pi/12*i, 3*math.pi/4)
    for i in range(3)
]
```

And plot the robot in each state, using `ipywidgets` (docs) to let us plot ganja side-by-side. Unfortunately, `pyganja` provides no mechanism to animate these plots from python.

This will not render side-by-side in the online clifford documentation, but will in a local notebook.

---

[7]:
```
import ipywidgets
from IPython.display import Latex, display
from pyganja import draw

outputs = [
    ipywidgets.Output(layout=ipywidgets.Layout(flex='1'))
    for i in range(len(theta_traj))
]
for output, thetas in zip(outputs, theta_traj):
    with output:
        # interesting part here - run the forward kinematics, print the angles we used
        serial_robot.thetas = thetas
        display(Latex(r"$\theta_i = {:.2f}, {:.2f}, {:.2f}$".format(*thetas)))
        draw(serial_robot.to_scene(), scale=1.5)
ipywidgets.HBox(outputs)
```

$\theta_i = 0.52, 1.05, 2.36$

```
<IPython.core.display.Javascript object>
```

$\theta_i = 0.79, 0.79, 2.36$

```
<IPython.core.display.Javascript object>
```

$\theta_i = 1.05, 0.52, 2.36$

```
<IPython.core.display.Javascript object>
```

```
HBox(children=(Output(layout=Layout(flex='1')), Output(layout=Layout(flex='1')),␣
↪Output(layout=Layout(flex='1'...
```

Inverse kinematics

(slides)

For the forward kinematics, we didn't actually need conformal geometric algebra at all—PGA would have done just fine, as all we needed were rotations and translations. The inverse kinematics of a serial manipulator is where CGA provide some nice tricks.

There are three facts we know about the position X, each of which describes a constraint surface

- X must lie on a sphere with radius l centered at Y, which can be written

$$S^* = Y - \frac{1}{2} l^2 n_\infty$$

- X must lie on a sphere with radius ρ centered at n_o, which can be written

$$S^*_{\text{base}} = n_0 - \frac{1}{2} \rho^2 n_\infty$$

- X must lie on a plane through n_o, e_3, and Y, which can be written

$$\Pi = n_0 \wedge \text{up}(e_3) \wedge Y \wedge n_\infty$$

Note that $\Pi = 0$ is possible iff $Y = \text{up}(k e_3)$.

For X to satisfy all three constraints. we have

$$S \wedge X = S_{\text{base}} \wedge X = \Pi \wedge X = 0 \tag{11.16}$$

$$X \wedge (\underbrace{S \vee S_{\text{base}} \vee \Pi}_{P}) = 0 \quad \text{If } \Pi \neq 0 \tag{11.17}$$

$$X \wedge (\underbrace{S \vee S_{\text{base}}}_{C}) = 0 \quad \text{otherwise} \tag{11.18}$$

$$\tag{11.19}$$

By looking at the grade of the term labelled P, we conclude it must be a point-pair—which tells us X must lie in one of two locations. Similarly, C must be a circle.

[8]:
```
@SerialRobot._add_method
def _get_x_constraints_for(self, Y):
    """ Get the space containing all possible elbow positions """
    # strictly should be undual, but we don't have that in clifford
    S = (Y - 0.5*self.l**2*einf).dual()
    S_base = (eo - 0.5*self.rho**2*einf).dual()
    Pi = eo ^ up(e2) ^ Y ^ einf
    return S, S_base, Pi

@SerialRobot._add_method
def _get_x_positions_for(self, Y):
    """ Get the space containing all possible elbow positions """
    S, S_base, Pi = self._get_x_constraints_for(Y)
    if Pi == 0:
        # any solution on the circle is OK
        return S & S_base
    else:
        # there are just two solutions
        return S & S_base & Pi
```

From the pointpair P we can extract the two possible X locations with:

$$X = \left[1 \pm \frac{P}{\sqrt{P\tilde{P}}}\right](P \cdot n_\infty)$$

To be considered a full solution to the inverse kinematics problem, we need to produce the angles $\theta_0, \theta_1, \theta_2$. We can do this as follows

[9]:
```
@SerialRobot._add_method
@SerialRobot.y_pos.setter
def y_pos(self, Y):
    R = generate_rotation_rotor
    T = generate_translation_rotor

    rotors = {}
    rotors[Links.UPPER, Links.ELBOW] = T(self.rho * e1)
    rotors[Links.FOREARM, Links.ENDPOINT] = T(-self.l * e1)

    x_options = self._get_x_positions_for(Y)
    if x_options.grades == {3}:
        # no need to adjust the base angle
        theta_0 = self.thetas[0]
        rotors[Links.BASE, Links.SHOULDER] = self.rotors[Links.BASE, Links.SHOULDER]
        # remove the rotation from x, intersect it with the plane of the links
        x_options = x_options & (eo ^ up(e3) ^ up(e1) ^ einf)
    else:
        y_down = down(Y)
        theta0 = math.atan2(y_down[(3,)], y_down[(1,)])
        rotors[Links.BASE, Links.SHOULDER] = R(theta0, e1, e3)

        # remove the first rotor from x
        x_options = apply_rotor(x_options, ~rotors[Links.BASE, Links.SHOULDER])

    # project out one end of the point-pair
    x = (1 - x_options.normal()) * (x_options | einf)

    x_down = down(x)
    theta1 = math.atan2(x_down[(2,)], x_down[(1,)])
    rotors[Links.SHOULDER, Links.UPPER] = R(theta1, e1, e2)

    _update_chain(rotors, Links.BASE, Links.SHOULDER, Links.UPPER)
    _update_chain(rotors, Links.BASE, Links.UPPER, Links.ELBOW)

    # remove the second rotor
    Y = apply_rotor(Y, ~rotors[Links.BASE, Links.ELBOW])
    y_down = down(Y)

    theta2 = math.atan2(-y_down[(2,)], -y_down[(1,)])
    rotors[Links.ELBOW, Links.FOREARM] = R(theta2, e1, e2)
    _update_chain(rotors, Links.BASE, Links.ELBOW, Links.FOREARM)
    _update_chain(rotors, Links.BASE, Links.FOREARM, Links.ENDPOINT)

    self._thetas = (theta0, theta1, theta2)
    self.rotors = rotors
```

Define a trajectory again, this time with a scene to render it:

[10]:
```
y_traj = [
    up(0.3*e3 + 0.8*e2 - 0.25*e1),
    up(0.6*e3 + 0.8*e2),
    up(0.9*e3 + 0.8*e2 + 0.25*e1)
]

expected_scene = GanjaScene()
expected_scene.add_facet(y_traj[0:2], color=(255, 128, 128))
expected_scene.add_facet(y_traj[1:3], color=(255, 128, 128))
```

And we can run the inverse kinematics by setting `serial_robot.y_pos`:

[11]:
```
outputs = [
    ipywidgets.Output(layout=ipywidgets.Layout(flex='1'))
    for i in range(len(y_traj))
]
first = True
for output, y in zip(outputs, y_traj):
    with output:
        # interesting part here - run the reverse kinematics, print the angles we used
        serial_robot.y_pos = y
        display(Latex(r"$\theta_i = {:.2f}, {:.2f}, {:.2f}$".format(*serial_robot.
↪thetas)))
        sc = serial_robot.to_scene()

        # Show the spheres we used to construct the solution
        sc += expected_scene
        if first:
            extra_scene = GanjaScene()
            S, S_base, Pi = serial_robot._get_x_constraints_for(y)
            extra_scene.add_object(S_base, label='S_base', color=(255, 255, 128))
            extra_scene.add_object(S, label='S', color=(255, 128, 128))
            extra_scene.add_object(Pi, label='Pi', color=(128, 255, 192, 128))
            sc += extra_scene
        draw(sc, scale=1.5)
    first = False
ipywidgets.HBox(outputs)
```

$\theta_i = 2.27, 1.64, 1.10$

```
<IPython.core.display.Javascript object>
```

$\theta_i = 1.57, 1.43, 1.32$

```
<IPython.core.display.Javascript object>
```

$\theta_i = 1.30, 1.11, 1.84$

```
<IPython.core.display.Javascript object>
```

```
HBox(children=(Output(layout=Layout(flex='1')), Output(layout=Layout(flex='1')),␣
↪Output(layout=Layout(flex='1'...
```

Parallel manipulators

For now, refer to the presentation

(slides)

Inverse kinematics

(slides)

For now, refer to the presentation

Forward kinematics

(slides)

For now, refer to the presentation

..... doc/tutorials/cga/robotic-manipulators.ipynb ends here.

..... doc/tutorials/cga/index.ipynb ends here.

The following section was generated from docs/tutorials/linear-transformations.ipynb .....

CHAPTER

TWELVE

LINEAR TRANSFORMATIONS

When working in regular vector spaces, a common tool is a linear transformation, typically in the form of a matrix.

While geometric algebra already provides the rotors as a means of describing transformations (see *the CGA tutorial section*), there are types of linear transformation that are not suitable for this representation.

This tutorial leans heavily on the explanation of linear transformations in GA4CS, chapter 4. It explores the *clifford.transformations* submodule.

12.1 Vector transformations in linear algebra

As a brief reminder, we can represent transforms in $\mathbb{R}^3$ using the matrices in $\mathbb{R}^{3\times3}$:

[1]:
```
import numpy as np

rot_and_scale_x = np.array([
    [1, 0, 0],
    [0, 1, -1],
    [0, 1, 1],
])
```

We can read this as a table, where each column corresponds to a component of the input vector, and each row a component of the output:

[2]:
```
def show_table(data, cols, rows):
    # trick to get a nice looking table in a notebook
    import pandas as pd; return pd.DataFrame(data, columns=cols, index=rows)
```

[3]:
```
show_table(rot_and_scale_x, ["$\mathit{in}_%s$" % c for c in "xyz"], ["$\mathit{out}_
↪%s$" % c for c in "xyz"])
```

[3]:

	in_x	in_y	in_z
out_x	1	0	0
out_y	0	1	−1
out_z	0	1	1

We can apply it to some vectors using the `@` matrix multiply operator:

[4]:
```
v1 = np.array([1, 0, 0])
v2 = np.array([0, 1, 0])
v3 = np.array([0, 0, 1])

(
```

(continues on next page)

(continued from previous page)

```
    rot_and_scale_x @ v1,
    rot_and_scale_x @ v2,
    rot_and_scale_x @ v3,
)
```

```
[4]: (array([1, 0, 0]), array([0, 1, 1]), array([ 0, -1,  1]))
```

We say this transformation is linear because $f(a+b) = f(a) + f(b)$:

```
[5]: assert np.array_equal(
    rot_and_scale_x @ (2*v1 + 3*v2),
    2 * (rot_and_scale_x @ v1) + 3 * (rot_and_scale_x @ v2)
)
```

12.2 Multivector transformations in geometric algebra

How would we go about applying `rot_and_scale_x` in a geometric algebra? Clearly we can apply it to vectors in the same way as before, which we can do by unpacking coefficients and repacking them:

```
[6]: from clifford.g3 import *

v = 2*e1 + 3*e2
v_trans = layout.MultiVector()
v_trans[1,], v_trans[2,], v_trans[3,] = rot_and_scale_x @ [v[1,], v[2,], v[3,]]
v_trans
```

```
[6]: (2.0^e1) + (3.0^e2) + (3.0^e3)
```

However, in geometric algebra we don't only care about the vectors, we want to transform the the higher-order blades too. This can be done via an outermorphism, which extends $f(a)$ to $f(a \wedge b) = f(a) \wedge f(b)$. This is where the `clifford.transformations` submodule comes in handy:

```
[7]: from clifford import transformations

rot_and_scale_x_ga = transformations.OutermorphismMatrix(rot_and_scale_x, layout)
```

To apply these transformations, we use the `()` operator, rather than `@`:

```
[8]: rot_and_scale_x_ga(e12)
```

```
[8]: (1^e12) + (1^e13)
```

```
[9]: # check it's an outermorphism
rot_and_scale_x_ga(e1) ^ rot_and_scale_x_ga(e2)
```

```
[9]: (1^e12) + (1^e13)
```

It shouldn't come as a surprise that applying the transformation to the psuedoscalar will tell us the determinant of our original matrix - the determinant tells us how a transformation scales volumes, and `layout.I` is a representation of the unit volume element!

```
[10]: np.linalg.det(rot_and_scale_x), rot_and_scale_x_ga(layout.I)
```

```
[10]: (2.0, (2^e123))
```

12.2.1 Matrix representation

Under the hood, clifford implements this using a matrix too - it's just now a matrix operating over all of the basis blades, not just over the vectors. We can see this by looking at the *private* `_matrix` attribute:

[11]:
```
show_table(rot_and_scale_x_ga._matrix, ["$\mathit{in}_{%s}$" % c for c in layout.
↪names], ["$\mathit{out}_{%s}$" % c for c in layout.names])
```

[11]:

	$\mathit{in}_{}$	in_{e1}	\
$\mathit{out}_{}$	1	0	
out_{e1}	0	1	
out_{e2}	0	0	
out_{e3}	0	0	
out_{e12}	0	0	
out_{e13}	0	0	
out_{e23}	0	0	
out_{e123}	0	0	

	in_{e2}	in_{e3}	\
$\mathit{out}_{}$	0	0	
out_{e1}	0	0	
out_{e2}	1	-1	
out_{e3}	1	1	
out_{e12}	0	0	
out_{e13}	0	0	
out_{e23}	0	0	
out_{e123}	0	0	

	in_{e12}	in_{e13}	\
$\mathit{out}_{}$	0	0	
out_{e1}	0	0	
out_{e2}	0	0	
out_{e3}	0	0	
out_{e12}	1	-1	
out_{e13}	1	1	
out_{e23}	0	0	
out_{e123}	0	0	

	in_{e23}	in_{e123}
$\mathit{out}_{}$	0	0
out_{e1}	0	0
out_{e2}	0	0
out_{e3}	0	0
out_{e12}	0	0
out_{e13}	0	0
out_{e23}	2	0
out_{e123}	0	2

..... doc/tutorials/linear-transformations.ipynb ends here.

The following section was generated from docs/tutorials/apollonius-cga-augmented.ipynb .....

CHAPTER

THIRTEEN

WORKING WITH CUSTOM ALGEBRAS

This notebook explores the algebra defined in The Lie Model for Euclidean Geometry (Hongbo Li), and its application to solving Apollonius' Problem. It also shows

The algebra is constructed with basis elements $e_{-2}, e_{-1}, e_1, \cdots, e_n, e_{n+1}$, where $e_{-2}^2 = -e_{-1}^2 = -e_{n+1}^2 = 1$. This is an extension of a standard conformal algebra, with an extra e_{n+1} basis vector.

Note that we permuted the order in the source code below to make `ConformalLayout` happy.

[1]:
```
from clifford import ConformalLayout, BasisVectorIds, MultiVector, transformations

class OurCustomLayout(ConformalLayout):
    def __init__(self, ndims):
        self.ndims = ndims

        euclidean_vectors = [str(i + 1) for i in range(ndims)]
        conformal_vectors = ['m2', 'm1']

        # Construct our custom algebra. Note that ConformalLayout requires the e- and
↪e+ basis vectors to be last.
        ConformalLayout.__init__(
            self,
            [1]*ndims + [-1] + [1, -1],
            ids=BasisVectorIds(euclidean_vectors + ['np1'] + conformal_vectors)
        )
        self.enp1 = self.basis_vectors_lst[ndims]

        # Construct a base algebra without the extra `enp1`, which would not be
↪understood by pyganja.
        self.conformal_base = ConformalLayout(
            [1]*ndims + [1, -1],
            ids=BasisVectorIds(euclidean_vectors + conformal_vectors)
        )

        # this lets us convert between the two layouts
        self.to_conformal = transformations.between_basis_vectors(self, self.
↪conformal_base)
```

The code above also defines a stardard conformal $\mathbb{R}^{N+1,1}$ layout without this new basis vector. This is primarily to support rendering with `pyganja`, which doesn't support the presence of this extra vector. `BasisVectorMap` defaults to preserving vectors by name between one algebra and another, while throwing away blades containing vectors missing from the destination algebra.

We define an `ups` function which maps conformal dual-spheres into this algebra, as $s' = s + |s|\, e_{n+1}$, and a `downs` that applies the correct sign. The `s` suffix here is chosen to mean `sphere`.

[2]:
```
def ups(self, s):
    return s + self.enp1*abs(s)

OurCustomLayout.ups = ups; del ups

def downs(self, mv):
    if (mv | self.enp1)[()] > 0:
        mv = -mv
    return mv

OurCustomLayout.downs = downs; del downs
```

Before we start looking at specified dimensions of euclidean space, we build a helper to construct conformal dual circles and spheres, with the word `round` being a general term intended to cover both circles and spheres.

[3]:
```
def dual_round(at, r):
    l = at.layout
    return l.up(at) - 0.5*l.einf*r*r
```

In order to render with `pyganja`, we'll need a helper to convert from our custom $\mathbb{R}^{N+1,2}$ layout into a standard conformal $\mathbb{R}^{N+1,1}$ layout. `clifford` maps indices in `.value` to basis blades via `layout._basis_blade_order.index_to_bitmap`, which we can use to convert the indices in one layout to the indices in another.

13.1 Visualization

Finally, we'll define a plotting function, which plots the problem and solution circles in suitable colors via `pyganja`. Note that this all works because of our definition of the `to_conformal` `BasisVectorMap`.

[4]:
```
import itertools
from pyganja import GanjaScene, draw

def plot_rounds(in_rounds, out_rounds, scale=1):
    colors = itertools.cycle([
        (255, 0, 0),
        (0, 255, 0),
        (0, 0, 255),
        (0, 255, 255),
    ])
    # note: .dual() neede here because we're passing in dual rounds, but ganja␣
↪expects direct rounds
    s = GanjaScene()
    for r, color in zip(in_rounds, colors):
        s.add_object(r.layout.to_conformal(r).dual(), color=color)
    for r in out_rounds:
        s.add_object(r.layout.to_conformal(r).dual(), color=(64, 64, 64))
    draw(s, sig=r.layout.conformal_base.sig, scale=scale)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/clifford/envs/stable/lib/python3.8/
↪site-packages/pyganja/__init__.py:2: UserWarning: Failed to import cef_gui, cef␣
↪functions will be unavailable
  from .script_api import *
```

13.2 Apollonius' problem in $\mathbb{R}^2$ with circles

[5]:
```
l2 = OurCustomLayout(ndims=2)
e1, e2 = l2.basis_vectors_lst[:2]
```

This gives us the `Layout l2` with the desired metric,

[6]:
```
import pandas as pd  # convenient but somewhat slow trick for showing tables
pd.DataFrame(l2.metric, index=l2.basis_names, columns=l2.basis_names)
```

[6]:

	e1	e2	enp1	em2	em1
e1	1.0	0.0	0.0	0.0	0.0
e2	0.0	1.0	0.0	0.0	0.0
enp1	0.0	0.0	-1.0	0.0	0.0
em2	0.0	0.0	0.0	1.0	0.0
em1	0.0	0.0	0.0	0.0	-1.0

Now we can build some dual circles:

[7]:
```
# add minus signs before `dual_round` to flip circle directions
c1 = dual_round(-e1-e2, 1)
c2 = dual_round(e1-e2, 0.75)
c3 = dual_round(e2, 0.5)
```

Compute the space orthogonal to all of them, which is an object of grade 2:

[8]:
```
pp = (l2.ups(c1) ^ l2.ups(c2) ^ l2.ups(c3)).dual()
pp.grades()
```

[8]:
```
{2}
```

We hypothesize that this object is of the form `l2.ups(c4) ^ l2.ups(c5)`. Taking a step not mentioned in the original paper, we decide to treat this as a regular conformal point pair, which allows us to project out the two factors with the approach taken in A Covariant Approach to Geometry using Geometric Algebra. Here, we normalize with e_{n+1} instead of the usual n_∞:

[9]:
```
def pp_ends(pp):
    P = (1 + pp.normal()) / 2
    return P * (pp | pp.layout.enp1), ~P * (pp | pp.layout.enp1)

c4u, c5u = pp_ends(pp)
```

And finally, plot our circles:

[10]:
```
plot_rounds([c1, c2, c3], [l2.downs(c4u), l2.downs(c5u)], scale=0.75)
```

```
<IPython.core.display.Javascript object>
```

This works for colinear circles too:

[11]:
```
c1 = dual_round(-1.5*e1, 0.5)
c2 = dual_round(e1*0, 0.5)
c3 = dual_round(1.5*e1, 0.5)
c4u, c5u = pp_ends((l2.ups(c1) ^ l2.ups(c2) ^ l2.ups(c3)).dual())

plot_rounds([c1, c2, c3], [l2.downs(c4u), l2.downs(c5u)])
```

```
<IPython.core.display.Javascript object>
```

[12]:
```
c1 = dual_round(-3*e1, 1.5)
c2 = dual_round(-2*e1, 1)
c3 = -dual_round(2*e1, 1)
c4u, c5u = pp_ends((l2.ups(c1) ^ l2.ups(c2) ^ l2.ups(c3)).dual())

plot_rounds([c1, c2, c3], [l2.downs(c4u), l2.downs(c5u)])
```

```
<IPython.core.display.Javascript object>
```

13.3 Apollonius' problem in $\mathbb{R}^3$ with spheres

[13]:
```
l3 = OurCustomLayout(ndims=3)
e1, e2, e3 = l3.basis_vectors_lst[:3]
```

Again, we can check the metric:

[14]:
```
pd.DataFrame(l3.metric, index=l3.basis_names, columns=l3.basis_names)
```

[14]:

	e1	e2	e3	enp1	em2	em1
e1	1.0	0.0	0.0	0.0	0.0	0.0
e2	0.0	1.0	0.0	0.0	0.0	0.0
e3	0.0	0.0	1.0	0.0	0.0	0.0
enp1	0.0	0.0	0.0	-1.0	0.0	0.0
em2	0.0	0.0	0.0	0.0	1.0	0.0
em1	0.0	0.0	0.0	0.0	0.0	-1.0

And apply the solution to some spheres, noting that we now need 4 in order to constrain our solution

[15]:
```
c1 = dual_round(e1+e2+e3, 1)
c2 = dual_round(-e1+e2-e3, 0.25)
c3 = dual_round(e1-e2-e3, 0.5)
c4 = dual_round(-e1-e2+e3, 1)
c5u, c6u = pp_ends((l3.ups(c1) ^ l3.ups(c2) ^ l3.ups(c3) ^ l3.ups(c4)).dual())

plot_rounds([c1, c2, c3, c4], [l3.downs(c6u), l3.downs(c5u)], scale=0.25)
```

```
<IPython.core.display.Javascript object>
```

Note that the figure above can be rotated!

doc/tutorials/apollonius-cga-augmented.ipynb ends here.

CHAPTER

FOURTEEN

OTHER RESOURCES FOR `CLIFFORD`

14.1 Slide Decks

- Installation
- Conformal Geometric Algebra with Python, Part 1
- Conformal Geometric Algebra with Python, Part 2

14.2 Videos

- Intro to Clifford

CHAPTER

FIFTEEN

OTHER RESOURCES FOR GEOMETRIC ALGEBRA

If you think Geometric Algebra looks interesting and want to learn more, check out these websites and textbooks

15.1 Links

- `galgebra`, a symbolic geometric algebra module for Python
- The Cambridge University Geometric Algebra Research Group home page
- David Hestenes' Geometric Calculus R & D Home Page

15.2 Introductory textbooks

- Geometric Algebra for Physicists, by Doran and Lasenby
- Geometric Algebra for Computer Science, by Dorst, Fontijne and Mann
- New Foundations for Classical Mechanics, by David Hestenes

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

X