MultiVectors

Release 1.2.1

Ken Hilton

Dec 25, 2022

CONTENTS:

CHAPTER

ONE

QUICK INTRO TO GEOMETRIC ALGEBRA

Here are some concepts to bear in mind. This is a *very* brief introduction to geometric algebra; a longer one is here.

1.1 Bases and geometric products

- Every dimension of space comes with a *basis vector*: an arrow of length 1 unit pointed towards the positive end of the axis.
  - In our 3-dimensional world, there are the basis vectors $\hat{x}$, $\hat{y}$, and $\hat{z}$, which are pointed towards the positive ends of the x, y, and z axes respectively.
  - The fourth dimensional basis vector is $\hat{w}$. In higher dimensions, usually all bases are numbered instead of lettered: the fifth dimensional basis vectors are $\hat{e}_1$, $\hat{e}_2$, $\hat{e}_3$, $\hat{e}_4$, and $\hat{e}_5$.
- The *geometric product* of two basis vectors is their simple multiplication - not the dot or cross product! The geometric product of $\hat{x}$ and $\hat{y}$ is simply $\hat{x}\hat{y}$.
  - The geometric product of two basis vectors is a *basis plane*. $\hat{x}\hat{y}$ is the basis plane of the xy plane. The other basis planes are $\hat{y}\hat{z}$ and $\hat{x}\hat{z}$.
  - The geometric product of three basis vectors is a *basis volume*. $\hat{x}\hat{y}\hat{z}$ is the basis volume of 3D space, which only has one basis volume, but also one of the four basis volumes of 4D space.
- The geometric product of a basis vector with itself is 1. That is, $\hat{x}\hat{x} = \hat{x}^2 = \hat{y}\hat{y} = \hat{y}^2 = \hat{z}\hat{z} = \hat{z}^2 = 1$.
- The geometric product of different basis vectors *anticommutes*: $\hat{x}\hat{y} = -\hat{y}\hat{x}$ and $\hat{x}\hat{y}\hat{z} = -\hat{x}\hat{z}\hat{y} = \hat{z}\hat{x}\hat{y} = -\hat{z}\hat{y}\hat{x}$.

1.2 Blades and multivectors

- A *blade* is a *scaled basis*: a *scalar* (regular real number) multiplied by a *basis*. For example, $3\hat{x}\hat{y}$ is a blade. Note that this means all bases are blades scaled by 1.
  - A k-*blade* is a blade of *grade* k: the geometric product of a scalar and k different basis vectors. $3\hat{x}\hat{y}$ has grade 2; it is a 2-blade.
  - Scalars are 0-blades - blades consisting of *no* basis vectors.
- A *multivector* is a sum of multiple blades. For example, $1 + 2\hat{x} - 3\hat{y}\hat{z}$ is a multivector.
  - The sum of multiple (and only) 1-blades is usually called a simple *vector*. For example, $3\hat{x} + 2\hat{y}$ is a vector.
  - The sum of multiple (and only) 2-blades is a *bivector*. Basis planes are also known as *basis bivectors*. For example, $3\hat{x}\hat{y}$ is a bivector.

- The rules of linearity, associativity and distributivity in multiplication apply, as long as order of arguments is maintained:
  - $(\hat{x})(a\hat{y}) = a\hat{x}\hat{y}$ (linearity, for scalar a)
  - $(\hat{x}\hat{y})(\hat{z}) = \hat{x}(\hat{y}\hat{z})$ (associativity)
  - $\hat{x}(\hat{y} + \hat{z}) = \hat{x}\hat{y} + \hat{x}\hat{z}$ (distributivity)
  - $(\hat{y} + \hat{z})(a\hat{x}) = a(\hat{y} + \hat{z})(\hat{x})$ (linearity) $= a(\hat{y}\hat{x} + \hat{z}\hat{x})$ (distributivity) $= a(-\hat{x}\hat{y} - \hat{x}\hat{z})$ (anticommutativity) $= -a(\hat{x}\hat{y} + \hat{x}\hat{z})$ (converse of distributivity)
- However, some things which require commutativity break down, such as the binomial theorem.

1.3 The choose operator, inner (dot) and outer (wedge) products

- $\langle V\rangle_n$ *chooses* all n-blades from the multivector V. For example, if $V = 1+2\hat{x}+3\hat{y}+4\hat{x}\hat{y}+5\hat{y}\hat{z}$, then $\langle V\rangle_0 = 1$ and $\langle V\rangle_1 = 2\hat{x} + 3\hat{y}$ and $\langle V\rangle_2 = 4\hat{x}\hat{y} + 5\hat{y}\hat{z}$.
- $U \cdot V = \langle UV\rangle_n$ where U is of grade r, V is of grade s, and $n = |r - s|$. This is the *inner* or *dot product*.
  - The dot product associates and distributes the same way the geometric product does.
  - From this, for arbitrary vectors $a\hat{x} + b\hat{y}$ and $c\hat{x} + d\hat{y}$, we recover the typical meaning of the dot product:

$$
\begin{aligned}
&(a\hat{x} + b\hat{y}) \cdot (c\hat{x} + d\hat{y}) \\
&= ac(\hat{x} \cdot \hat{x}) + ad(\hat{x} \cdot \hat{y}) + bc(\hat{y} \cdot \hat{x}) + bd(\hat{y} \cdot \hat{y}) \\
&= ac\langle\hat{x}\hat{x}\rangle_0 + ad\langle\hat{x}\hat{y}\rangle_0 + bc\langle\hat{y}\hat{x}\rangle_0 + bd\langle\hat{y}\hat{y}\rangle_0 \\
&= ac\langle 1\rangle_0 + ad(0) + bc(0) + bd\langle 1\rangle_0 \\
&\text{(because } \hat{x}\hat{y} \text{ and } \hat{y}\hat{z} \text{ have no part with grade } 0) \\
&= ac + bd
\end{aligned}
$$

- $U \wedge V = \langle UV\rangle_n$ where U is of grade r, V is of grade s, and $n = r + s$. This is the *outer* or *wedge product*.
  - The outer product associates and distributes the same way the geometric product does.
  - From this, for arbitrary vectors $a\hat{x} + b\hat{y} + c\hat{z}$ and $d\hat{x} + e\hat{y} + f\hat{z}$, we recover something that looks very much like a cross product:

$$
\begin{aligned}
&(a\hat{x} + b\hat{y} + c\hat{z}) \wedge (d\hat{x} + e\hat{y} + f\hat{z}) \\
&= ad(\hat{x} \wedge \hat{x}) + bd(\hat{y} \wedge \hat{x}) + cd(\hat{z} \wedge \hat{x}) \\
&\quad + ae(\hat{x} \wedge \hat{y}) + be(\hat{y} \wedge \hat{y}) + ce(\hat{z} \wedge \hat{y}) \\
&\quad + af(\hat{x} \wedge \hat{z}) + bf(\hat{y} \wedge \hat{z}) + cf(\hat{z} \wedge \hat{z}) \\
&= ad\langle\hat{x}\hat{x}\rangle_2 + be\langle\hat{y}\hat{y}\rangle_2 + cf\langle\hat{z}\hat{z}\rangle_2 \\
&\quad + (ae - bd)\langle\hat{x}\hat{y}\rangle_2 + (af - cd)\langle\hat{x}\hat{z}\rangle_2 + (bf - ce)\langle\hat{y}\hat{z}\rangle_2 \\
&= 0 + 0 + 0 + \begin{vmatrix} a & b \\ d & e \end{vmatrix}(\hat{x}\hat{y}) + \begin{vmatrix} a & c \\ d & f \end{vmatrix}(\hat{x}\hat{z}) + \begin{vmatrix} b & c \\ e & f \end{vmatrix}(\hat{y}\hat{z}) \\
&\text{(because } \hat{x}\hat{x} \text{ etc.} = 1\text{, which has no grade 2 part)} \\
&= \begin{vmatrix} \hat{y}\hat{z} & \hat{x}\hat{z} & \hat{x}\hat{y} \\ a & b & c \\ d & e & f \end{vmatrix}
\end{aligned}
$$

1.4 Euler's formula applied to multivectors

- $e^{\theta B} = \cos\theta + B\sin\theta$ where θ is a scalar in radians and B is a basis multivector.

Note: This formula only works when $B^2 = -1$ (in the same way as the imaginary unit i), such as $(\hat{x}\hat{y})^2 = \hat{x}\hat{y}\hat{x}\hat{y} = -\hat{x}\hat{y}\hat{y}\hat{x} = -\hat{x}\hat{x} = -1$.

  – For purposes of interest, the more general formula for e raised to a multivector power is the Taylor series:

$$e^V = \exp(V) = \sum_{n=0}^{\infty} \frac{V^n}{n!}$$

- For reasons that are beyond my power to explain, the rotation of a multivector V by θ through the plane B is $e^{-\frac{\theta B}{2}} V e^{\frac{\theta B}{2}}$.

CHAPTER

TWO

AS APPLIED IN THIS LIBRARY

All of the concepts in *Quick Intro to Geometric Algebra* are applied in this library.

2.1 Blades

- `multivectors.x`, `y`, `z`, and `w` represent the first four basis vectors. Bases past 4D use $\hat{e}_n$ syntax: `multivectors.e5` or `multivectors.e_5` represent the 5th basis vector.
- Basis names can be *swizzled* on the module:

```
>>> from multivectors import x, y, z, xyz
>>> x * y * z == xyz
True
```

- `multivectors._` is a 0-blade: a scalar, but with `MultiVector` type. It is differentiated from normal scalars when `repr()`-d by surrounding parentheses:

```
>>> from multivectors import _
>>> 2 * _
(2.0)
```

- The rules of arithmetic with blades as described above apply:

```
>>> from multivectors import x, y, z
>>> x * 2 + x * 3
(5.0 * x)
>>> x * 5 * y
(5.0 * x*y)
>>> (x * y) * z == x * (y * z)
True
```

2.2 MultiVector

- A multivector is a sum of zero or more blades. All scalars and blades are multivectors, but not all multivectors are scalars or blades.
- Basis names can be swizzled on class **instances** to get the coefficient of that basis:

```
>>> from multivectors import x, y
>>> (x + 2*y).x
1.0
```

- Basis indices can also be used:

```
>>> from multivectors import xy, yz
>>> (xy + 2*yz) % (0, 1)
1.0
```

- Choosing by grade is supported:

```
>>> from multivectors import x, y, xy, yz
>>> V = 1 + 2*x + 3*y + 4*xy + 5*yz
>>> V[0]
(1.0)
>>> V[1]
(2.0 * x + 3.0 * y)
>>> V[2]
(4.0 * x*y + 5.0 * y*z)
```

- Getting the grade of a multivector will only work if if the multivector is a blade. Otherwise, you get `None`:

```
>>> from multivectors import w, xy, yz
>>> w.grade
1
>>> xy.grade
2
>>> yz.grade
2
>>> (w + xy).grade
>>> # returns None
>>> (xy + yz).grade
>>> # even multivectors consisting of blades with the same grade don't work
>>> # the multivector must be an actual blade, with only one basis sequence
```

- The rules of arithmetic with multivectors as described above apply:

```
>>> from multivectors import x, y, z
>>> x * (y + z)
(1.0 * x*y + 1.0 * x*z)
>>> (y + z) * (2 * x)
(-2.0 * x*y + -2.0 * x*z)
>>> (1*x + 2*y) * (3*x + 4*y)
(11.0 + -2.0 * x*y)
```

- The extra products apply too:

```
>>> from multivectors import x, y, z
>>> 1*3 + 2*4
11
>>> (1*x + 2*y) @ (3*x + 4*y)
(11.0)
>>> (1*5 - 2*4, 1*6 - 3*4, 2*6 - 3*5)
(-3, -6, -3)
>>> (1*x + 2*y + 3*z) ^ (4*x + 5*y + 6*z)
(-3.0 * x*y + -6.0 * x*z + -3.0 * y*z)
```

- A convenience method is provided to rotate multivectors:

```
>>> from math import radians
>>> from multivectors import x, y, z, xz
>>> round((3*x + 2*y + 4*z).rotate(radians(90), xz), 2)
(-4.0 * x + 2.0 * y + 3.0 * z)
```

CHAPTER

THREE

MULTIVECTORS MODULE REFERENCE

3.1 Module Attributes

`multivectors._`

The scalar basis multivector (a 0-blade). Usage:

```
>>> from multivectors import _
>>> 2 * _
(2.0)
```

`multivectors.x`

`multivectors.y`

`multivectors.z`

`multivectors.w`

Basis vectors for the first four dimensions. Additional bases can be swizzled on the module:

```
>>> from multivectors import x, y, z, xyz
>>> x * y * z == xyz
True
```

3.2 The MultiVector class

class `multivectors.`**`MultiVector`**(*termdict: Dict[Tuple[int, ...], float]*)

A linear combination of geometric products of basis vectors.

The bare constructor is not meant for regular use. Use module swizzling or the factory *`from_terms()`* instead.

Basis vector names can be swizzled on instances:

```
>>> from multivectors import x, y, z
>>> (x + y).x
1.0
>>> (x*y + z).xy
1.0
>>> (x + y).e3
0.0
```

And indices can be combined:

```
>>> (x + y) % 0
1.0
>>> (x*y + z) % (0, 1)
1.0
>>> (x + y) % 2
0.0
```

property grade: Optional[int]

The grade of this blade.

Returns

The number of different bases this blade consists of, or None if this multivector is not a blade (one term).

Examples

```
>>> from multivectors import x, y, z
>>> (x + y).grade # not a blade
>>> (z * 2 + z).grade
1
>>> (x*y*z).grade
3
```

property terms: Tuple[*MultiVector*, ...]

Get a sequence of blades comprising this multivector.

Examples

```
>>> from multivectors import x, y, z, w
>>> (x + y).terms
((1.0 * x), (1.0 * y))
>>> ((x + y) * (z + w)).terms
((1.0 * x*z), (1.0 * x*w), (1.0 * y*z), (1.0 * y*w))
```

classmethod from_terms(**terms: Union[float, MultiVector]*) → *MultiVector*

Create a multivector by summing a sequence of terms.

Parameters

***terms** – The terms. If you have an iterable of terms, use (e.g.) `from_terms(*terms)`

Returns

A multivector.

Examples

```
>>> from multivectors import x, y, z
>>> MultiVector.from_terms(x, y)
(1.0 * x + 1.0 * y)
>>> MultiVector.from_terms()
(0.0)
>>> MultiVector.from_terms(z)
(1.0 * z)
>>> MultiVector.from_terms(2 * x, x)
(3.0 * x)
```

classmethod scalar(*num*) → *MultiVector*

Create a MultiVector representing a scalar.

Parameters
`num` – Any object that can be `float()`-d.

Returns
A multivector with only a scalar part of `num`.

Examples

```
>>> from multivectors import MultiVector
>>> MultiVector.scalar('0')
(0.0)
>>> MultiVector.scalar('1.2')
(1.2)
```

__getattr__(*name: str*) → float

Support basis name swizzling.

__getitem__(*grades: Union[int, Iterable[int], slice]*) → *MultiVector*

The choose operator - returns the sum of all blades of grade k.

Parameters
`grades` – The grade(s) to choose.

Examples

```
>>> from multivectors import x, y, z
>>> (1 + 2*x + 3*y + 4*x*y)[1]
(2.0 * x + 3.0 * y)
>>> (1 + 2 + 3*x + 4*x*y + 5*y*z)[2]
(4.0 * x*y + 5.0 * y*z)
>>> (1 + 2 + 3*x + 4*x*y + 5*y*z)[0]
(3.0)
>>> (1 + 2 + 3*x + 4*x*y + 5*y*z).choose(0)
(3.0)
>>> (1 + 2*x + 3*x*y)[:2]
(1.0 + 2.0 * x)
>>> (1 + 2*x + 3*x*y)[1:]
(2.0 * x + 3.0 * x*y)
```

choose(*grades: Union[int, Iterable[int], slice]*) → *MultiVector*

The choose operator - returns the sum of all blades of grade k.

Parameters

grades – The grade(s) to choose.

Examples

```
>>> from multivectors import x, y, z
>>> (1 + 2*x + 3*y + 4*x*y)[1]
(2.0 * x + 3.0 * y)
>>> (1 + 2 + 3*x + 4*x*y + 5*y*z)[2]
(4.0 * x*y + 5.0 * y*z)
>>> (1 + 2 + 3*x + 4*x*y + 5*y*z)[0]
(3.0)
>>> (1 + 2 + 3*x + 4*x*y + 5*y*z).choose(0)
(3.0)
>>> (1 + 2*x + 3*x*y)[:2]
(1.0 + 2.0 * x)
>>> (1 + 2*x + 3*x*y)[1:]
(2.0 * x + 3.0 * x*y)
```

__repr__() → str

Return a representation of this multivector. Depending on the global namespace, this may be `eval()`-able.

Examples

```
>>> from multivectors import x, y, z, w
>>> repr(x)
'(1.0 * x)'
>>> repr(x + y)
'(1.0 * x + 1.0 * y)'
>>> repr(y*z - x*w)
'(-1.0 * x*w + 1.0 * y*z)'
```

__str__() → str

Return a representation of this multivector suited for showing.

Examples

```
>>> from multivectors import x, y, z, w
>>> str(x)
'1.00x'
>>> str(x + y)
'(1.00x + 1.00y)'
>>> str(y*z - x*w)
'(-1.00xw + 1.00yz)'
>>> print(1 + x + x*y)
(1.00 + 1.00x + 1.00xy)
```

__format__(*spec: str*) → str

Return a representation of this multivector suited for formatting.

Parameters

spec – The format spec (forwarded to the underlying :class:`float`s)

Examples

```
>>> from multivectors import x, y, z, w
>>> f'{x:.3f}'
'1.000x'
>>> V = x + 2 * y + z
>>> '{:.3f}'.format(V)
'(1.000x + 2.000y + 1.000z)'
>>> V = y*z - x*w
>>> f'{V:.1f}'
'(-1.0xw + 1.0yz)'
```

__eq__(*other: Union[float, MultiVector]*) → bool

Compare equality of two objects.

Returns

`True` if all terms of this multivector are equal to the `other`; `True` if this multivector is scalar and equals the `other`; or `False` for all other cases or types.

Examples

```
>>> from multivectors import x, y
>>> x + y == y + x
True
>>> x + 2*y == 2*x + y
False
```

__ne__(*other: Union[float, MultiVector]*) → bool

Compare inequality of two objects.

Returns

`False` if all terms of this multivector are equal to the `other`; `False` if this multivector is scalar and equals the `other`; or `True` for all other cases or types.

Examples

```
>>> from multivectors import x, y
>>> x + y != y + x
False
>>> x + 2*y != 2*x + y
True
```

__lt__(*other: float*) → bool

Compare this blade less than an object.

Returns
True if this is a scalar blade less than the scalar; False if this is a scalar blade not less than the scalar; or NotImplemented for all other types.

Examples

```
>>> from multivectors import _, x
>>> _ * 1 < 2
True
>>> _ * 2 < 1
False
>>> x * 1 < 2
Traceback (most recent call last):
    ...
TypeError: '<' not supported between instances of 'MultiVector' and 'int'
```

__gt__(*other: float*) → bool
Compare this blade greater than an object.

Returns
True if this is a scalar blade greater than the scalar; False if this is a scalar blade not greater than the scalar; or NotImplemented for all other types.

Examples

```
>>> from multivectors import _, x
>>> _ * 1 > 2
False
>>> _ * 2 > 1
True
>>> x * 1 > 2
Traceback (most recent call last):
    ...
TypeError: '>' not supported between instances of 'MultiVector' and 'int'
```

__le__(*other: float*) → bool
Compare this blade less than or equal to an object.

Returns
True if this is a scalar blade less than or equal to the scalar; False if this is a scalar blade greater than the scalar; or NotImplemented for all other types.

Examples

```
>>> from multivectors import _, x
>>> _ * 1 <= 2
True
>>> _ * 2 <= 2
True
>>> _ * 2 <= 1
False
```

(continues on next page)

(continued from previous page)

```
>>> x * 1 <= 2
Traceback (most recent call last):
    ...
TypeError: '<=' not supported between instances of 'MultiVector' and 'int'
```

__ge__(*other: float*) → bool

Compare this blade greater than or equal to an object.

Returns
`True` if this is a scalar blade greater than or equal to the scalar; `False` if this is a scalar blade less than the scalar; or `NotImplemented` for all other types.

Examples

```
>>> from multivectors import _, x
>>> _ * 1 >= 2
False
>>> _ * 2 >= 2
True
>>> _ * 2 >= 1
True
>>> x * 1 >= 2
Traceback (most recent call last):
    ...
TypeError: '>=' not supported between instances of 'MultiVector' and 'int'
```

__add__(*other: Union[float, MultiVector]*) → *MultiVector*

Add a multivector and another object.

Examples

```
>>> from multivectors import x, y, z, w
>>> (x + z) + (y + w)
(1.0 * x + 1.0 * y + 1.0 * z + 1.0 * w)
>>> (x + z) + y
(1.0 * x + 1.0 * y + 1.0 * z)
>>> (x + y) + 1
(1.0 + 1.0 * x + 1.0 * y)
```

__radd__(*other: float*) → *MultiVector*

Support adding multivectors on the right side of objects.

Examples

```
>>> from multivectors import x, y, z
>>> 1 + (x + z)
(1.0 + 1.0 * x + 1.0 * z)
>>> x + (y + z)
(1.0 * x + 1.0 * y + 1.0 * z)
```

__sub__(*other: Union[float, MultiVector]*) → *MultiVector*

Subtracting is adding the negation.

Examples

```
>>> from multivectors import x, y, z, w
>>> (x + z) - (y + w)
(1.0 * x + -1.0 * y + 1.0 * z + -1.0 * w)
>>> (x + z) - y
(1.0 * x + -1.0 * y + 1.0 * z)
```

__rsub__(*other: float*) → *MultiVector*

Support subtracting multivectors from objects.

Examples

```
>>> from multivectors import x, y, z, w
>>> 1 - (x + z)
(1.0 + -1.0 * x + -1.0 * z)
>>> x - (y + z)
(1.0 * x + -1.0 * y + -1.0 * z)
```

__mul__(*other: Union[float, MultiVector]*) → *MultiVector*

Multiply a multivector and another object.

Returns
`(a + b) * (c + d) = a*c + a*d + b*c + b*d` for multivectors `(a + b)` and `(c + d)`.

Returns
`(a + b) * v = a*v + b*v` for multivector `(a + b)` and scalar `v`.

Examples

```
>>> from multivectors import x, y, z, w
>>> (x + y) * (z + w)
(1.0 * x*z + 1.0 * x*w + 1.0 * y*z + 1.0 * y*w)
>>> (x + y) * 3
(3.0 * x + 3.0 * y)
>>> (x + y) * x
(1.0 + -1.0 * x*y)
```

__rmul__(*other: float*) → *MultiVector*

Support multiplying multivectors on the right side of scalars.

Examples

```
>>> from multivectors import x, y
>>> 3 * (x + y)
(3.0 * x + 3.0 * y)
>>> y * (x + y)
(1.0 + -1.0 * x*y)
```

__matmul__(*other: Union[float,* MultiVector*]*) → *MultiVector*

Get the inner (dot) product of two objects.

Returns
`u @ v = (u * v)[abs(u.grade - v.grade)]` when *grade* is defined.

Returns
`(a + b) @ (c + d) = a@c + a@d + b@c + b@d` for multivectors `(a + b)` and `(c + d)`

Returns
`(a + b) @ v = a@v + b@v` for multivector `(a + b)` and scalar v

Examples

```
>>> from multivectors import x, y, z
>>> (2*x + 3*y) @ (4*x + 5*y)
(23.0)
>>> (2*x*y).dot(3*y*z)
(0.0)
>>> (x + y).inner(3)
(3.0 * x + 3.0 * y)
>>> (x + y) @ x
(1.0)
```

dot(*other: Union[float,* MultiVector*]*) → *MultiVector*

Get the inner (dot) product of two objects.

Returns
`u @ v = (u * v)[abs(u.grade - v.grade)]` when *grade* is defined.

Returns
`(a + b) @ (c + d) = a@c + a@d + b@c + b@d` for multivectors `(a + b)` and `(c + d)`

Returns
`(a + b) @ v = a@v + b@v` for multivector `(a + b)` and scalar v

Examples

```
>>> from multivectors import x, y, z
>>> (2*x + 3*y) @ (4*x + 5*y)
(23.0)
>>> (2*x*y).dot(3*y*z)
(0.0)
>>> (x + y).inner(3)
(3.0 * x + 3.0 * y)
>>> (x + y) @ x
(1.0)
```

inner(*other: Union[float,* MultiVector*]*) → *MultiVector*

Get the inner (dot) product of two objects.

Returns
`u @ v = (u * v)[abs(u.grade - v.grade)]` when *grade* is defined.

Returns
`(a + b) @ (c + d) = a@c + a@d + b@c + b@d` for multivectors `(a + b)` and `(c + d)`

Returns
`(a + b) @ v = a@v + b@v` for multivector `(a + b)` and scalar `v`

Examples


```
>>> from multivectors import x, y, z
>>> (2*x + 3*y) @ (4*x + 5*y)
(23.0)
>>> (2*x*y).dot(3*y*z)
(0.0)
>>> (x + y).inner(3)
(3.0 * x + 3.0 * y)
>>> (x + y) @ x
(1.0)
```


__rmatmul__(*other: float*) → *MultiVector*

Support dotting multivectors on the right hand side.

Returns
`v @ (a + b) = v@a + v@b` for multivector `(a + b)` and scalar `v`

Examples

```
>>> from multivectors import x, y
>>> 3 @ (x + y)
(3.0 * x + 3.0 * y)
>>> x @ (x + y)
(1.0)
```

__truediv__(*other: Union[float,* MultiVector*]*) → *MultiVector*

Divide two objects.

Returns
`(a + b) / v = a/v + b/v`

Examples

```
>>> from multivectors import x, y
>>> (6*x + 9*y) / 3
(2.0 * x + 3.0 * y)
>>> (6*x + 9*y) / (3*x)
(2.0 + -3.0 * x*y)
```

`__rtruediv__`(*other: float*) → *MultiVector*

Divide a scalar by a multivector. Only defined for blades.

Examples

```
>>> from multivectors import x, y
>>> 1 / x
(1.0 * x)
>>> 2 / (4 * x*y)
(-0.5 * x*y)
```

`__mod__`(*idxs: Union[int, Iterable[int], slice]*) → float

Support index swizzling.

Examples

```
>>> from multivectors import x, y
>>> (x + y) % 0
1.0
>>> v = 1 + 2*y + 3*x*y
>>> v % ()
1.0
>>> v % 1
2.0
>>> v % (0, 1)
3.0
>>> v % 2
0.0
```

`__pow__`(*other: int*) → *MultiVector*

A multivector raised to an integer power.

`V ** n = V * V * V * ... * V`, n times. `V ** -n = 1 / (V ** n)`

Examples

```
>>> from multivectors import x, y
>>> (x + y) ** 3
(2.0 * x + 2.0 * y)
>>> (2 * x*y) ** -5
(-0.03125 * x*y)
```

__rpow__(*other: float*) → *MultiVector*

A real number raised to a multivector power.

`x ** V = e ** ln(x ** V) = e ** (V ln x)`

Examples

```
>>> from multivectors import x, y
>>> round(2 ** (x + y), 2)
(1.52 + 0.81 * x + 0.81 * y)
```

exp() → *MultiVector*

e raised to this multivector power.

$$e^V = \exp(V) = \sum_{n=0}^{\infty} \frac{V^n}{n!}$$

Examples

```
>>> from math import pi, sqrt
>>> from multivectors import x, y
>>> # 45-degree rotation through xy-plane
>>> # results in (1+xy)/sqrt(2)
>>> (pi/4 * x*y).exp() * sqrt(2)
(1.0 + 1.0 * x*y)
>>> round((pi * x*y).exp(), 14)
(-1.0)
```

__xor__(*other: Union[float, MultiVector]*) → *MultiVector*

Get the outer (wedge) product of two objects.

> **Warning:** Operator precedence puts `^` after `+`! Make sure to put outer products in parentheses, like this: `u * v == u @ v + (u ^ v)`

Returns
`u ^ v = (u * v)[u.grade + v.grade]` when *grade* is defined

Returns
`(a + b) ^ (c + d) = (a^c) + (a^d) + (b^c) + (b^d)` for multivector `(a + b)` and `(c + d)`

Returns
`(a + b) ^ v = (a^v) + (b^v)` for multivector `(a + b)` and scalar v

Examples

```
>>> from multivectors import x, y, z
>>> (2*x + 3*y) ^ (4*x + 5*y)
(-2.0 * x*y)
>>> (2*x*y).wedge(3*y*z)
(0.0)
>>> (x + y).outer(3)
(3.0 * x + 3.0 * y)
>>> (x + y) ^ x
(-1.0 * x*y)
```

wedge(*other: Union[float,* MultiVector*]*) → *MultiVector*

Get the outer (wedge) product of two objects.

> **Warning:** Operator precedence puts ^ after +! Make sure to put outer products in parentheses, like this: `u * v == u @ v + (u ^ v)`

Returns
`u ^ v = (u * v)[u.grade + v.grade]` when *grade* is defined

Returns
`(a + b) ^ (c + d) = (a^c) + (a^d) + (b^c) + (b^d)` for multivector `(a + b)` and `(c + d)`

Returns
`(a + b) ^ v = (a^v) + (b^v)` for multivector `(a + b)` and scalar v

Examples


```
>>> from multivectors import x, y, z
>>> (2*x + 3*y) ^ (4*x + 5*y)
(-2.0 * x*y)
>>> (2*x*y).wedge(3*y*z)
(0.0)
>>> (x + y).outer(3)
(3.0 * x + 3.0 * y)
>>> (x + y) ^ x
(-1.0 * x*y)
```


outer(*other: Union[float,* MultiVector*]*) → *MultiVector*

Get the outer (wedge) product of two objects.

> **Warning:** Operator precedence puts ^ after +! Make sure to put outer products in parentheses, like this: `u * v == u @ v + (u ^ v)`

Returns
`u ^ v = (u * v)[u.grade + v.grade]` when *grade* is defined

Returns
`(a + b) ^ (c + d) = (a^c) + (a^d) + (b^c) + (b^d)` for multivector `(a + b)` and `(c + d)`

Returns
`(a + b) ^ v = (a^v) + (b^v)` for multivector `(a + b)` and scalar `v`

Examples

```
>>> from multivectors import x, y, z
>>> (2*x + 3*y) ^ (4*x + 5*y)
(-2.0 * x*y)
>>> (2*x*y).wedge(3*y*z)
(0.0)
>>> (x + y).outer(3)
(3.0 * x + 3.0 * y)
>>> (x + y) ^ x
(-1.0 * x*y)
```

__rxor__(*other: float*) → *MultiVector*

Support wedging multivectors on the right hand side.

Returns
`v ^ (a + b) = (v^a) + (v^b)` for multivector `(a + b)` and simple `v`

Examples

```
>>> from multivectors import x, y
>>> 3 ^ (x + y)
(3.0 * x + 3.0 * y)
>>> x ^ (x + y)
(1.0 * x*y)
```

__neg__() → *MultiVector*

The negation of a multivector is the negation of all its terms.

Examples

```
>>> from multivectors import x, y, z
>>> -(2*x + 3*y*z)
(-2.0 * x + -3.0 * y*z)
```

__pos__() → *MultiVector*

A normalized multivector is one scaled down by its magnitude.

Examples

```
>>> from multivectors import x, y, z, w
>>> +(x + y + z + w)
(0.5 * x + 0.5 * y + 0.5 * z + 0.5 * w)
>>> round((x + y).normalize(), 3)
(0.707 * x + 0.707 * y)
```

normalize() → *MultiVector*

A normalized multivector is one scaled down by its magnitude.

Examples

```
>>> from multivectors import x, y, z, w
>>> +(x + y + z + w)
(0.5 * x + 0.5 * y + 0.5 * z + 0.5 * w)
>>> round((x + y).normalize(), 3)
(0.707 * x + 0.707 * y)
```

__abs__() → float

The magnitude of a multivector is the square root of the sum of the squares of its terms.

Examples

```
>>> from multivectors import x, y, z, w
>>> abs(x + y + z + w)
2.0
>>> (2 ** 1.5 * x + 2 ** 1.5 * y).magnitude()
4.0
```

magnitude() → float

The magnitude of a multivector is the square root of the sum of the squares of its terms.

Examples

```
>>> from multivectors import x, y, z, w
>>> abs(x + y + z + w)
2.0
>>> (2 ** 1.5 * x + 2 ** 1.5 * y).magnitude()
4.0
```

__invert__() → *MultiVector*

The conjugate of a multivector is the negation of all terms besides the real component.

Examples

```
>>> from multivectors import x, y
>>> ~(1 + 2*x + 3*x*y)
(1.0 + -2.0 * x + -3.0 * x*y)
>>> (2 + 3*y).conjugate()
(2.0 + -3.0 * y)
```

conjugate() → *MultiVector*

The conjugate of a multivector is the negation of all terms besides the real component.

Examples

```
>>> from multivectors import x, y
>>> ~(1 + 2*x + 3*x*y)
(1.0 + -2.0 * x + -3.0 * x*y)
>>> (2 + 3*y).conjugate()
(2.0 + -3.0 * y)
```

__complex__() → complex

Convert a scalar to a complex number.

Examples

```
>>> from multivectors import _, x
>>> complex(2 * _)
(2+0j)
>>> complex(3 * x)
Traceback (most recent call last):
    ...
TypeError: cannot convert non-scalar blade (3.0 * x) to complex
```

__int__() → int

Convert a scalar to an integer.

Examples

```
>>> from multivectors import _, x
>>> int(2 * _)
2
>>> int(3 * x)
Traceback (most recent call last):
    ...
TypeError: cannot convert non-scalar blade (3.0 * x) to int
```

__float__() → float

Convert a scalar to a float.

Examples

```
>>> from multivectors import _, x
>>> float(2 * _)
2.0
>>> float(3 * x)
Traceback (most recent call last):
    ...
TypeError: cannot convert non-scalar blade (3.0 * x) to float
```

__round__(*ndigits: Optional[int] = None*) → *MultiVector*

Round the scalars of each component term of a multivector.

Examples

```
>>> from multivectors import x, y
>>> round(1.7 * x + 1.2 * y)
(2.0 * x + 1.0 * y)
>>> round(0.15 * x + 0.05 * y, 1)
(0.1 * x + 0.1 * y)
```

__trunc__() → *MultiVector*

Truncate the scalars of each component term of a multivector.

Examples

```
>>> import math
>>> from multivectors import x, y
>>> math.trunc(1.7 * x + 1.2 * y)
(1.0 * x + 1.0 * y)
>>> math.trunc(-1.7 * x - 1.2 * y)
(-1.0 * x + -1.0 * y)
```

__floor__() → *MultiVector*

Floor the scalars of each component term of a multivector.

Examples

```
>>> import math
>>> from multivectors import x, y
>>> math.floor(1.7 * x + 1.2 * y)
(1.0 * x + 1.0 * y)
>>> math.floor(-1.7 * x - 1.2 * y)
(-2.0 * x + -2.0 * y)
```

__ceil__() → *MultiVector*

Ceiling the scalars of each component term of a multivector.

Examples

```
>>> import math
>>> from multivectors import x, y
>>> math.ceil(1.7 * x + 1.2 * y)
(2.0 * x + 2.0 * y)
>>> math.ceil(-1.7 * x - 1.2 * y)
(-1.0 * x + -1.0 * y)
```

rotate(*angle: float*, *plane:* MultiVector) → *MultiVector*

Rotate this multivector by angle in rads around the blade plane.

Parameters

- **angle** – Angle to rotate by, in radians.
- **plane** – Blade representing basis plane to rotate through.

Returns

Rotated multivector.

Examples

```
>>> from math import radians
>>> from multivectors import x, y, z, w
>>> round((3*x + 2*y + 4*z).rotate(
...     radians(90), x*y), 2)
(-2.0 * x + 3.0 * y + 4.0 * z)
>>> round((3*x + 2*y + 4*z + 5*w).rotate(
...     radians(90), x*y*z), 2)
(3.0 * x + 2.0 * y + 4.0 * z + -5.0 * x*y*z*w)
```

angle_to(*other:* MultiVector) → float

Get the angle between this multivector and another.

Examples

```
>>> from math import degrees
>>> from multivectors import x, y, z, w
>>> math.degrees((x + y).angle_to(x - y))
90.0
>>> round(math.degrees((x + y + z + w).angle_to(x - y - z - w)), 2)
120.0
```

__hash__ = None

3.3 Helper Functions

These functions are not really meant for exporting, but are included for completeness.

`multivectors.merge`(*arr: List[int]*, *left: List[int]*, *right: List[int]*) → int

Perform a merge of sorted lists and count the swaps.

Parameters

- **arr** – The list to write into.
- **left** – The left sorted list.
- **right** – The right sorted list.

Returns

The number of swaps made when merging.

`multivectors.count_swaps`(*arr: List[int]*, *copy: bool = True*) → int

Count the number of swaps needed to sort a list.

Parameters

- **arr** – The list to sort.
- **copy** – If `True`, (the default) don't modify the original list.

Returns

The number of swaps made when sorting the list.

Examples

```
>>> count_swaps([1, 3, 2, 5, 4])
2
>>> count_swaps([3, 2, 1])
3
```

`multivectors.names_to_idxs`(*name: str*, *raise_on_invalid_chars: bool = False*) → List[int]

Convert swizzled basis vector names into generalized basis indices.

Parameters

- **name** – The names to convert.
- **raise_on_invalid_chars** – If `True` (default `False`), raise `AttributeError` if any characters appear in the name that are invalid in a basis name.

Returns

A list of basis indexes that the name represents.

Raises

`AttributeError` – If characters invalid for a basis name appear, and `raise_on_invalid_chars` is `True`.

Examples

```
>>> names_to_idxs('xyw')
[0, 1, 3]
>>> names_to_idxs('e_1e2_z')
[0, 1, 2]
>>> names_to_idxs('_')
[]
```

`multivectors.`**`idxs_to_idxs`**(*idxs: Union[int, Iterable[int], slice]*) → List[int]

Convert multiple possible ways to specify multiple indices.

This is intended to be given the argument to *`MultiVector`* indexing: `V[0]` would call `idxs_to_idxs(0)`, `V[0, 1]` would call `idxs_to_idxs((0, 1))`, and `V[0:1]` would call `idxs_to_idxs(slice(0, 1, None))`

Parameters

idxs – The indexes to convert. This can be an integer, for just that index; an iterable of integers, for those indexes directly; or a slice, for the indexes the slice represents.

Returns

A list of basis indexes, converted from the argument.

Examples

```
>>> idxs_to_idxs(slice(None, 5, None))
[0, 1, 2, 3, 4]
>>> idxs_to_idxs((1, 3, 4))
[1, 3, 4]
>>> idxs_to_idxs(1)
[1]
```

`multivectors.`**`idxs_to_names`**(*idxs: Union[int, Iterable[int], slice], sep: str = ''*) → str

Convert indices to a swizzled name combination.

Parameters

- **idxs** – The basis index(es), as accepted by *`idxs_to_idxs()`*.
- **sep** – The separator to use between the basis vector names.

Returns

The swizzled names.

Examples

```
>>> idxs_to_names(slice(None, 5, None))
'e1e2e3e4e5'
>>> idxs_to_names((0, 1, 3))
'xyw'
>>> idxs_to_names(2)
'z'
>>> idxs_to_names((0, 2, 3), sep='*')
'x*z*w'
```

multivectors.**condense_bases**(*bases: Tuple[int, ...], scalar: float = 1.0*) → Tuple[Tuple[int, ...], float]

Normalize a sequence of bases, modifying the scalar as necessary.

Parameters

- **bases** – The tuple of basis indices.
- **scalar** – Real number that will scale the resulting bases.

Returns

A 2-tuple of normalized bases and the modified scalar.

Examples

```
>>> condense_bases((1, 1, 2, 1, 2), 2.0)
((1,), -2.0)
>>> condense_bases((1, 2, 1, 2), 1.5)
((), -1.5)
>>> condense_bases((2, 1, 3, 2, 3, 3), 1.0)
((1, 3), 1.0)
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

C

D

E

F

G

I

M

N

O

R

S

T

W

X

Y

Z