The Wedge, AND, OR and XOR Products

Kurt Nalty

March 27, 2017

Abstract

Hermann Grassmann's wedge product has been widely adopted in physics and mathematics. However, the related regressive product is rarely presented, due to varying interpretations and prescriptions for implementing the regressive product. Instead, various methods are presented for unions and intersections, joins and meets.

This note presents a simple set of products based upon AND, OR and XOR logic operations operating on geometric elements.

The Wedge Product in Three Dimensions

The wedge product in three dimensions is defined by the basis multiplication table shown below. The scalar element is q, the linear vector elements are x y and z, the bivector elements are xy, zx and yz, while the pseudovector is xyz.

Multiplication table format using Wedge, prefactor on left

	PostFactor							
	q	x	y	z	xy	zx	yz	xyz
q	q	x	y	z	xy	zx	yz	xyz
x	x	0	xy	-zx	0	0	xyz	0
y	y	-xy	0	yz	0	xyz	0	0
z	z	zx	-yz	0	xyz	0	0	0
xy	xy	0	0	xyz	0	0	0	0
zx	zx	0	xyz	0	0	0	0	0
yz	yz	xyz	0	0	0	0	0	0
xyz	xyz	0	0	0	0	0	0	0

In the wedge product, any repeated basis products lead to zero result. This product preserves or raised the grades of the terms involved as in x*yz = xyz.

Criticism of Grassmann's Implementation

Authors such as John Browne and translator Lloyd C. Kannenberg reject Grassmann's implementation of the regressive product as merging the scalar and psuedoscalar terms, leading to difficulties with units and change of coordinates similar to vector/pseudo-vector confusion in regular vector math. Grassmann wanted to combine the wedge and regressive products in a fashion akin to Clifford algebra. Browne, Kannenberg and others insist on maintaining separate products.

The Regressive Product DeMorgan Form

John Browne defines his regressive product in terms of a complement operation coupled with the wedge product. His complement operation is finding the terms necessary to make a pseudoscalar basis product for that term of interest.

Using an overbar to indicate complement, $\overline{x} = yz$, $\overline{xy} = z$, and so forth.

Given the complement operation, the regressive product is naively written as $A \vee B = \overline{\overline{A} \wedge \overline{B}}$, which is called out as the DeMorgan related form.

Eric Lengyel points out that there are really two complements, being right and left complements, which coincide for odd dimensions. Using overline for right complement, underline for left complement, we can define the antiwedge product via DeMorgan style duality relationships.

$$A \vee B = \underline{\overline{A} \wedge \overline{B}} = \overline{\underline{A} \wedge \underline{B}}$$

David Hestenes uses post-multiplication by the inverse of the pseudoscalar to define his complement. The resulting regressive product then is defined as

$$A \vee B = ((AI^{-1}) \wedge (BI^{-1}))I^{-1}$$

With each of these products, we quickly become aware of some fine print. We have to make some assumptions about the dimensionality of the space

involved when forming the complement. For example, in 3D space, $\overline{x} = yz$, while in 2D space, $\overline{x} = y$. Browne and Hestenes, for example, require the dimensionality of the space be reduced to the minimum which spans all product terms, and this dimensionality, in turns, defines the character of the complement. As an example, $x \vee xy$ will have a two dimensional space with pseudovector xy, while $xy \vee yz$ will have a three dimensional space with pseudovector xyz.

Legnyel, by contrast, suggests leaving the dimensionality fixed at the overall space, so that in three dimensions, the pseudovector is always xyz.

These types of discussions have led to defining different operations, join and meet, rather than worrying about the regressive product in its various forms.

The Regressive AND Product

I believe that there is a much easier way to define the regressive product, which focuses on the greatest common factor between two basis terms. Rather than explore space/anti-space duality, we instead look at the intersection between the two terms. We can see the product by inspection, regardless of dimensionality, as in $x \vee xy = x$. Being more formal, we can represent each basis by a bit in and binary number. The AND operation of the basis being multiplied is our representation for the product.

In tabular form, this multiplication can be represented in the table below. Notice, I am using the zx bivector format for this table. Also note that I use the scalar q, rather than assume the value zero for terms with nothing else in common.

Regressive AND Basis Product, zx format

	q	x	y	xy	z	zx	yz	xyz
q	q	q	q	q	q	q	q	q
x	q	x	q	x	q	x	q	x
y	q	q	y	y	q	q	y	y
xy	q	x	y	xy	q	x	y	xy
z	q	q	q	q	z	z	z	z
zx	q	x	q	x	z	zx	z	zx
yz	q	q	y	y	z	z	yz	yz
xyz	q	x	y	xy	z	zx	yz	xyz

Implementing this product in C++, with a zx format, results in a multivector Regressive_AND bit of code.

```
c.q   = + a.q*b.q + a.q*b.x + a.q*b.y + a.q*b.xy + a.q*b.z + a.q*b.zx
        + a.q*b.yz + a.q*b.xyz + a.x*b.q + a.x*b.y + a.x*b.z
        + a.x*b.yz + a.y*b.q + a.y*b.x + a.y*b.z + a.y*b.zx
        + a.xy*b.q + a.xy*b.z + a.z*b.q + a.z*b.x + a.z*b.y
        + a.z*b.xy + a.zx*b.q + a.zx*b.y + a.yz*b.q + a.yz*b.x
        + a.xyz*b.q  ;
c.x   = + a.x*b.x + a.x*b.xy + a.x*b.zx + a.x*b.xyz + a.xy*b.x
        + a.xy*b.zx + a.zx*b.x + a.zx*b.xy + a.xyz*b.x  ;
c.y   = + a.y*b.y + a.y*b.xy + a.y*b.yz + a.y*b.xyz + a.xy*b.y
        + a.xy*b.yz + a.yz*b.y + a.yz*b.xy + a.xyz*b.y  ;
c.xy  = + a.xy*b.xy + a.xy*b.xyz + a.xyz*b.xy ;
c.z   = + a.z*b.z + a.z*b.zx + a.z*b.yz + a.z*b.xyz + a.zx*b.z
        + a.zx*b.yz + a.yz*b.z + a.yz*b.zx + a.xyz*b.z  ;
c.zx  = + a.zx*b.zx + a.zx*b.xyz + a.xyz*b.zx ;
c.yz  = + a.yz*b.yz + a.yz*b.xyz + a.xyz*b.yz ;
c.xyz = + a.xyz*b.xyz;
```

The Regressive_AND, as defined above, is commutative and associative.

Interesting enough, I redid the product table, using xz format. The resulting formulas were also commutative and associative.

I have also toyed with simplifing the c.q terms. As long as the remaining terms are symmetrical between a and b, we maintain commutative and associate behavior. As an example, c.q = a.q*b.q or c.q = a.q*b.q + a.q*b.x + a.x*b.q .

The OR Product

Replacing the AND function by the OR function provides the OR product.

In tabular form, this multiplication can be represented in the table below. Notice, I am using the zx bivector format for this table. Also note that I use the scalar q, rather than assume the value zero for terms with nothing else in common.

OR Basis Product, zx format

	q	x	y	xy	z	zx	yz	xyz
q	q	x	y	xy	z	zx	yz	xyz
x	x	x	xy	xy	zx	zx	xyz	xyz
y	y	xy	y	xy	yz	xyz	yz	xyz
xy	xy	xy	xy	xy	xyz	xyz	xyz	xyz
z	z	zx	yz	xyz	z	zx	yz	xyz
zx	zx	zx	xyz	xyz	zx	zx	xyz	xyz
yz	yz	xyz	yz	xyz	yz	xyz	yz	xyz
xyz	xyz	xyz	xyz	xyz	xyz	xyz	xyz	xyz

Here is the OR product, implemented in this product in C++, with a zx format. This product is associative and commutative.

```
c.q   = + a.q*b.q  ;
c.x   = + a.q*b.x + a.x*b.q + a.x*b.x  ;
c.y   = + a.q*b.y + a.y*b.q + a.y*b.y  ;
c.xy  = + a.q*b.xy + a.x*b.y + a.x*b.xy + a.y*b.x + a.y*b.xy
        + a.xy*b.q + a.xy*b.x + a.xy*b.y + a.xy*b.xy ;
c.z   = + a.q*b.z + a.z*b.q + a.z*b.z  ;
c.zx  = + a.q*b.zx + a.x*b.z + a.x*b.zx + a.z*b.x + a.z*b.zx
        + a.zx*b.q + a.zx*b.x + a.zx*b.z + a.zx*b.zx ;
c.yz  = + a.q*b.yz + a.y*b.z + a.y*b.yz + a.z*b.y + a.z*b.yz
        + a.yz*b.q + a.yz*b.y + a.yz*b.z + a.yz*b.yz ;
c.xyz = + a.q*b.xyz + a.x*b.yz + a.x*b.xyz + a.y*b.zx + a.y*b.xyz
        + a.xy*b.z + a.xy*b.zx + a.xy*b.yz + a.xy*b.xyz + a.z*b.xy
        + a.z*b.xyz + a.zx*b.y + a.zx*b.xy + a.zx*b.yz + a.zx*b.xyz
        + a.yz*b.x + a.yz*b.xy + a.yz*b.zx + a.yz*b.xyz + a.xyz*b.q
        + a.xyz*b.x + a.xyz*b.y + a.xyz*b.xy + a.xyz*b.z + a.xyz*b.zx
        + a.xyz*b.yz + a.xyz*b.xyz ;
```

The XOR Product

In tabular form, this multiplication can be represented in the table below. Notice, I am using the zx bivector format for this table. Also note that I use the scalar q, rather than assume the value zero for terms with nothing else in common.

XOR Basis Product, zx format

	q	x	y	xy	z	zx	yz	xyz
q	q	x	y	xy	z	zx	yz	xyz
x	x	q	xy	y	zx	z	xyz	yz
y	y	xy	q	x	yz	xyz	z	zx
xy	xy	y	x	q	xyz	yz	zx	z
z	z	zx	yz	xyz	q	x	y	xy
zx	zx	z	xyz	yz	x	q	xy	y
yz	yz	xyz	z	zx	y	xy	q	x
xyz	xyz	yz	zx	z	xy	y	x	q

Here is the XOR product, implemented in this product in C++, with a zx format. This product is commutative and associative.

```
c.q   = + a.q*b.q + a.x*b.x + a.y*b.y + a.xy*b.xy
        + a.z*b.z + a.zx*b.zx + a.yz*b.yz + a.xyz*b.xyz ;
c.x   = + a.q*b.x + a.x*b.q + a.y*b.xy + a.xy*b.y
        + a.z*b.zx + a.zx*b.z + a.yz*b.xyz + a.xyz*b.yz ;
c.y   = + a.q*b.y + a.x*b.xy + a.y*b.q + a.xy*b.x
        + a.z*b.yz + a.zx*b.xyz + a.yz*b.z + a.xyz*b.zx ;
c.xy  = + a.q*b.xy + a.x*b.y + a.y*b.x + a.xy*b.q
        + a.z*b.xyz + a.zx*b.yz + a.yz*b.zx + a.xyz*b.z  ;
c.z   = + a.q*b.z + a.x*b.zx + a.y*b.yz + a.xy*b.xyz
        + a.z*b.q + a.zx*b.x + a.yz*b.y + a.xyz*b.xy ;
c.zx  = + a.q*b.zx + a.x*b.z + a.y*b.xyz + a.xy*b.yz
        + a.z*b.x + a.zx*b.q + a.yz*b.xy + a.xyz*b.y  ;
c.yz  = + a.q*b.yz + a.x*b.xyz + a.y*b.z + a.xy*b.zx
        + a.z*b.y + a.zx*b.xy + a.yz*b.q + a.xyz*b.x  ;
c.xyz = + a.q*b.xyz + a.x*b.yz + a.y*b.zx + a.xy*b.z
        + a.z*b.xy + a.zx*b.y + a.yz*b.x + a.xyz*b.q  ;
```

Comments

These products are clearly different from the conventional regressive product definitions. I am checking these products for join, meet and comparision applications.

References

[1] John Browne, *Grassmann Algebra*, Barnard Publishing, 9-781479-197637

[2] Hermann Grassmann, *A New Branch of Mathematics*, Translated by Lloyd C. Kannenberg, Open Court Publishing, ISBN 0-8126-9276-4

[3] Chris Doran and Anthony Lasenby, *Geometric Algebra for Physicists*, Cambridge University Press, ISBN 978-0-521-71595-9

[4] Leo Dorst, Daniel Fontune and Stephen Mann, *Geometric Algebra for Computer Science*, Morgan Kaufmann Publishers, ISBN 978-0-12-374942-0

[5] David Hestenes and Garret Sobczyk, *Clifford Algebra to Geometric Calculus*, D. Reidal Publishing Company, ISBN 978-90-277-2581-5

[6] Anthony Lasenby and Chris Doran, *Lectures and Handouts 1999*, `www.mrao.cam.ac.uk/clifford/ptIIIcourse/`

[7] Eric Lengyel, *Foundations of Game Engine Development*, Terathon Software LLC, ISBN 9-780985-811747