Geometric Algebra

7. Implementation

Dr Chris Doran
ARM Research

Implementation

1. What is the appropriate data structure?
2. How do we implement the geometric product?
3. Symbolic computation with Maple
4. Programming languages

Large array

`Type: [Float]`

Vectors in 3D $[0, a_1, a_2, a_3, 0, 0, 0, 0]$

Bivector in 4D $[0, 0, 0, 0, 0, E_1, E_2, E_3, B_1, B_2, B_3, 0, 0, 0, 0, 0]$

For	Against
• Arrays are a hardware friendly data structure • Objects are fairly strongly typed • Do not need a separate multiplication matrix for each type	• Very verbose and wasteful • Need to know the dimension of the space up front • Hard to move between dimensions • Need a separate implementation of the product for each dimension and signature

Compact array

`Type: [Float]`

Vectors in 3D $[E_1, E_2, E_3]$

Bivector in 3D $[B_1, B_2, B_3]$

For	Against
• Arrays are a compact data structure – hardware friendly • Most familiar • Difficult to imagine a more compact structure	• Objects are no-longer typed • Size of the space needed up front • Hard to move between dimensions • Separate implementation of the product for each dimension, signature and grade • Sum of different grades?

Intrinsic Representation

Vectors in 3D $[(a_1, a_2, a_3)]$

As a sum of blades `a1*e[1]+a2*e[2]+a3*e[3]`

For	Against
• Strongly typed • Dense • Only need to know how to multiply blade elements together • Multiplication is a map operator • Don’t need to know dimension of space…	• Relying on typography to encode blades, etc. • Still need to compile down to a more basic structure • Need a way to calculate basis blade products

Symbolic algebra

Range of Symbolic Algebra packages are available:

- Maple
- Mathematica
- Maxima
- SymPy

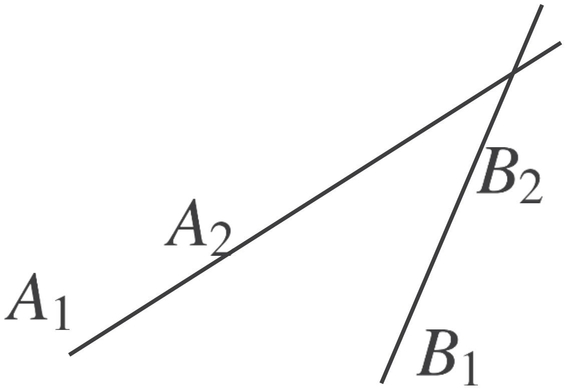
A good GA implementation for Maple has existed for 20 years: http://geometry.mrao.cam.ac.uk/2016/11/symbolic-algebra-and-ga/

- SA (Euclidean space)
- STA (Spacetime algebra)
- MSTA (Multiparticle STA)
- Default (e[i] has positive norm, and e[-i] has negative norm)
- Multivectors are built up symbolically or numerically
- Great for complex algebraic work (gauge theory gravity)

Examples

Intersection of two lines

$$A_1 = (1, 0)$$
$$A_2 = (2, 1)$$
$$B_1 = (4, 0)$$
$$B_2 = (5, 3)$$
$$L_a = A_1 \wedge A_2$$
$$L_b = B_1 \wedge B_2$$




```
res = 11*e[1]+9*e[2]+2*e[3]
```

$$R = (5.5, 4.5)$$

Case of parallel lines

```
res = -e[1]
```

Examples

```
vderiv2 := proc(mvin)
    local tx, ty, res;
    tx := diff(mvin,x);
    ty := diff(mvin,y);
    res := e[1]&@tx + e[2]&@ty;
end:
```

Maple procedure for 2d vector derivative for multivector function of x and y

Boosting a null vector:

```
n := e[0] + e[1];
res := psi&@nn&@reverse(psi)
4*e[0]+4*e[1]
```

GA Code

Want a representation where:

- Multivectors are encoded as dense lists
- We carry round the blade and coefficient together (in a tuple)
- We have a geometric product and a projection operator
- The geometric product works on the individual blades
- Ideally, do not multiply coefficients when result is not needed
- All expressed in a functional programming language

Why Haskell?

Functional

Functions are first-class citizens

- The can be passed around like variables
- Output of a function can be a function

Gives rise to the idea of higher-order functions

Functional languages are currently generating considerable interest:

- Haskell, Scala, ML, Ocaml, F#

Immutable data

(Nearly) all data is immutable: never change a variable

- Always create a new variable, then let garbage collector free up memory
- No messing around with pointers!

Linked lists are the natural data type

Why Haskell?

Purity

Functions are pure

- Always return same output for same input
- No side-effects

Natural match for scientific computing

Evaluations are thread-safe

Strong typing

Haskell is strongly typed, and statically typed

All code is checked for type integrity before compilation

- A lot of bugs are caught this way!

Strongly typed multivectors can remove ambiguity

- Are 4 numbers a quaternion?
- or a projective vector …

Why Haskell?

Recursion

Recursive definition of functions is compact and elegant
Supported by powerful pattern matching
Natural to mathematicians

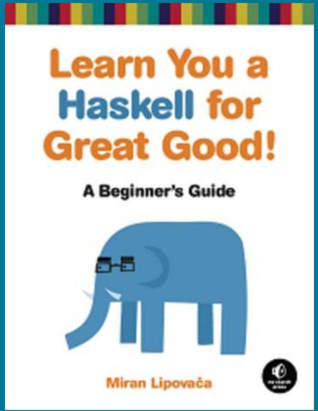
Laziness

Haskell employs lazy evaluation – call by need
Avoids redundant computation
Good match for GA

$$\langle AB \rangle_r$$

Higher-level code

GA is a higher-level language for mathematics
High-level code that is clear, fast and many-core friendly
Code precisely mirrors the mathematics
“Programming in GA”



learnyouahaskell.com
haskell.org/platform
wiki.haskell.org

Bit vector representation of blades

Details depend on whether you want to use mixed signature space

Best to stay as general as possible

Blade	Bit vector	Integer
1	0	0
e1	1	1
f1	01	2
e2	001	4
f2	0001	8
e1f1	11	3
e1e2	101	5

Geometric product is an xorr operation

Careful with typographical ordering here!

Have to take care of sign in geometric product

```
(Num a, Integral n) => (n,a)
```

Linked list

Type: `[(Int,Float)] or [(Blade)]`

Vectors in 3D $[(1, a_1), (4, a_2), (8, a_3)]$

As an ordered list `(1,a1):(2,a2):(8,a3):[]`

For	Against
• Strongly typed • Dense • Only need to know how to multiply blade elements together • Multiplication is a map operator • Don’t need to know dimension of space…	• Linked-lists are not always optimal • Depends how good the compiler is at managing lists in the cache • May need a look-up table to store blade products (though this is not always optimal)

Conversion functions

```
int2bin :: (Integral n) => n -> [Int]
int2bin 0 = [0]
int2bin 1 = [1]
int2bin n
    | even n = 0: int2bin (n `div` 2)
    | otherwise = 1: int2bin ((n-1) `div` 2)

bin2int :: (Integral n) => [Int] -> n
bin2int [0] = 0
bin2int [1] = 1
bin2int (x:xs)
    | x ==0 = 2 * (bin2int xs)
    | otherwise = 1 + 2 * (bin2int xs)
```

Note the recursive definition of these functions

A typical idiom in Haskell (and other FP languages)

These are other way round to typical binary

Currying

```
bladeGrade :: (Integral n) => n -> Int
bladeGrade = sum.int2bin
```

Suppress the argument in the function definition.

Haskell employs ‘currying’ – everything is a function with 1 variable.

Functions with more than one variable are broken down into functions that return functions

```
g :: (a,b) ->  c
f :: a ->  b -> c
f :: a-> (b -> c)
```

f takes in an argument and returns a new function

Blade product

```
bladeProd (n,a) (m,b) = (r,x)
   where (r,fn) = bldProd n m
         x = fn (a*b)
```

The `bldProd` function must (in current implementation)

1. Convert integers to bitvector rep
2. Compute the xorr and convert back to base 10
3. Add up number of sign changes from anticommutation
4. Add up number of sign changes from signature
5. Compute overall sign and return this

Can all be put into a LUT

Or use memoization

Candidate for hardware acceleration

Blade Product

```
bldProd :: (Integral n, Num a) => n -> n -> (n, a->a)
bldProd n m = ((bin2int (resBld nb mb)),fn)
  where nb = int2bin n
        mb = int2bin m
        tmp = ((countSwap nb mb) + (countNeg nb mb)) `mod` 2
        fn = if tmp == 0 then id else negate
```

Returns a function in second slot

Counts the number of swaps to bring things into normal order

Counts number of negative norm vectors that are squared

Geometric product

$$AB$$

`[Blades]` → AB ← `[Blades]`

```
A*B=simplify([bladeprod(a,b) | a <- A, b <- B])
```

Form every combination of product from the two lists

Sort by grade and then integer order

Combine common entries

Build up everything from

1. Multivector product
2. Projection onto grade
3. Reverse

Use * for multivector product

Abstract Data Type

```
newtype Multivector n a = Mv [(n,a)]
mv :: (Num a, Eq a) => [(a,String)] -> Multivector Int a
mv xs = Mv (bladeListSimp (sortBy bladeComp (map blade xs)))

longMv :: (Num a, Eq a) => [(a,String)] -> Multivector Integer a
longMv xs = Mv (bladeListSimp (sortBy bladeComp (map blade xs)))
```

Type class restrictions are put into the constructors.

Two constructors to allow for larger spaces (Int may only go up to 32D)

Class Membership

Want to belong to Num class

```
instance (Integral n, Num a, Eq a) => Num (Multivector n a) where
  (Mv xs) * (Mv ys) = Mv (bladeListProduct xs ys)
  (Mv xs) + (Mv ys) = Mv (bladeListAdd xs ys)
  fromInteger n = Mv [(0,fromInteger n)]
  negate (Mv xs) = Mv (bldListNegate xs)
  abs (Mv xs) = Mv xs
  signum (Mv xs) = Mv xs
```

Can now use + and * the way we would naturally like to!

Other resources (GA wikipedia page)

- GA Viewer Fontijne, Dorst, Bouma & Mann
  http://www.geometricalgebra.net/downloads.html
- Gaigen Fontijne. For programmers, this is a code generator with support for C, C++, C# and Java.
  http://www.geometricalgebra.net/new.html
- Gaalop Gaalop (Geometic Algebra Algorithms Optimizer) is a software to optimize geometric algebra files.
  http://www.gaalop.de/
- Versor, by Colapinto. A lightweight templated C++ Library with an OpenGL interface
  http://versor.mat.ucsb.edu/

Resources

geometry.mrao.cam.ac.uk
chris.doran@arm.com
cjld1@cam.ac.uk
@chrisjldoran
#geometricalgebra
github.com/ga