Geometric Algebra

8. Unification and Implementation

Dr Chris Doran
ARM Research

Euclidean geometry

Represent the Euclidean point *x* by null vectors

$$X = -\bar{n} + 2x + x^2 n$$

Distance is given by the inner product

$$\frac{-2X\cdot Y}{X\cdot n\, Y\cdot n} = (\boldsymbol{x} - \boldsymbol{y})^2$$

$$\frac{-X}{X\cdot n} = -\frac{1}{2}\bar{n} + x + \frac{1}{2}x^2 n$$

Read off the Euclidean vector

Depends on the concept of the origin

Spherical geometry

Suppose instead we form
$$\frac{-X}{X\cdot\bar{e}} = \hat{x} + \bar{e}$$

Unit vector in an n+1 dimensional space

Instead of plotting points in Euclidean space, we can plot them on a sphere

No need to pick out a preferred origin any more

$$\frac{-X\cdot Y}{X\cdot\bar{e}\,Y\cdot\bar{e}} = -(\hat{x}\cdot\hat{y} - 1)$$
$$= 2\sin^2(\theta/2)$$

Spherical geometry

Spherical distance $$d(\hat{x}, \hat{y}) = 2\sin^{-1}\left(\frac{-X\cdot Y}{2X\cdot\bar{e}\,Y\cdot\bar{e}}\right)^{1/2}$$

Same pattern as Euclidean case

'Straight' lines are now

$$X\wedge Y\wedge\bar{e} = \hat{x}\wedge\hat{y}\bar{e}$$

The $\bar{e}$ term now becomes essentially redundant and drops out of calculations

Invariance group are the set of rotors satisfying $R\bar{e}\tilde{R} = \bar{e}$

Generators satisfy

$$B\cdot\bar{e} = 0$$

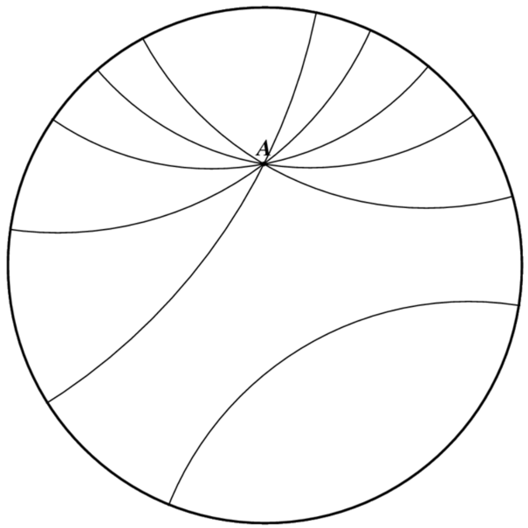
Left with standard rotors in a Euclidean space. Just rotate the unit sphere

non-Euclidean geometry

Historically arrived at by replacing the parallel postulate

‘Straight’ lines become d-lines. Intersect the unit circle at 90°

Model this in our conformal framework



Unit circle $e_1 e_2 \bar{e} = Ie$

d-lines $L \wedge e = 0$

d-line between *X* and *Y* is

$$L = X \wedge Y \wedge e$$

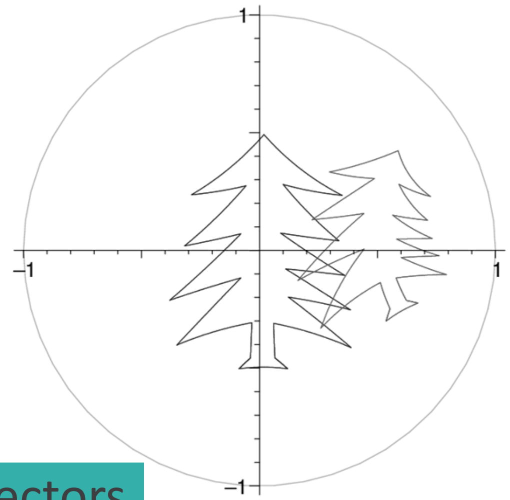
$$L^2 > 0$$

Translation along a d-line generated by

$$B = Le \quad B^2 > 0$$

Rotor generates hyperbolic transformations

non-Euclidean geometry



$$Y = e^{\alpha \hat{B}/2} X e^{-\alpha \hat{B}/2} \qquad \hat{B} = \frac{B}{|B|}$$

Generator of translation along the d-line.
Use this to define distance.

Write $X = \hat{x} + e, \quad Y = \hat{y} + e$

Unit time-like vectors

$$\hat{x} \cdot \hat{y} = \cosh(\alpha)$$

Boost factor from special relativity

$$\cosh(\alpha) = 1 - \frac{X \cdot Y}{X \cdot e \, Y \cdot e}$$

$$\sinh^2(\alpha/2) = -\frac{X \cdot Y}{2X \cdot e \, Y \cdot e}$$

$$d(x, y) = 2 \sinh^{-1}\left(-\frac{X \cdot Y}{2X \cdot e \, Y \cdot e}\right)^{1/2}$$

Distance in non-Euclidean geometry

non-Euclidean distance

$$d(x, y) = 2\sinh^{-1}\left(\frac{|x-y|^2}{(1-x^2)(1-y^2)}\right)^{1/2}$$

Distance expands as you get near to the boundary

Circle represents a set of points at infinity

This is the Poincare disk view of non-Euclidean geometry

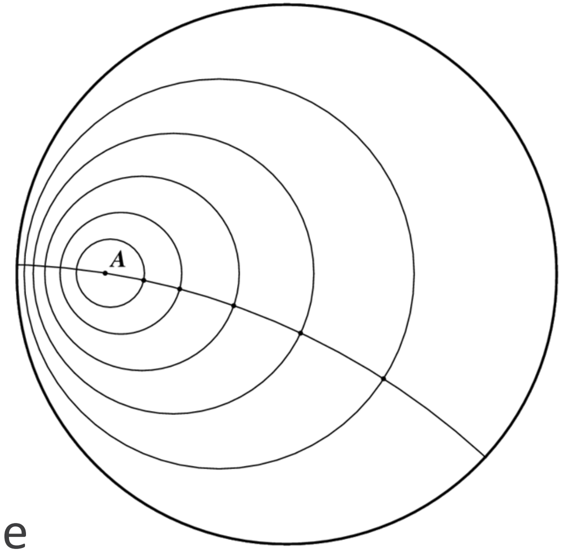
non-Euclidean circles

$$-\frac{X\cdot C}{2X\cdot e\, C\cdot e} = \text{constant} = \alpha^2$$

$$X\cdot(\underline{C + 2\alpha^2 C\cdot e\, e}) = 0$$

$$s = IS \qquad X\wedge S = 0$$

Formula unchanged from the Euclidean case



Still have $S = X_1\wedge X_2\wedge X_3$

Non-Euclidean circle

Definition of the centre is not so obvious. Euclidean centre is

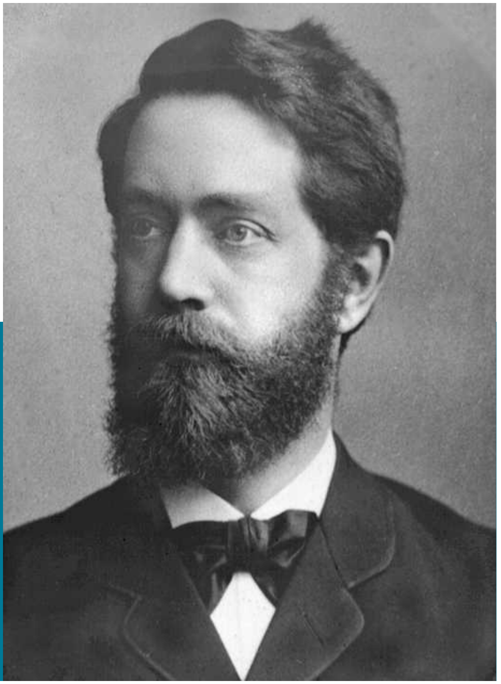
$$C = SnS$$

Reverse the logic above and define $C = s + \lambda e$

$$C^2 = 0 \implies \lambda$$

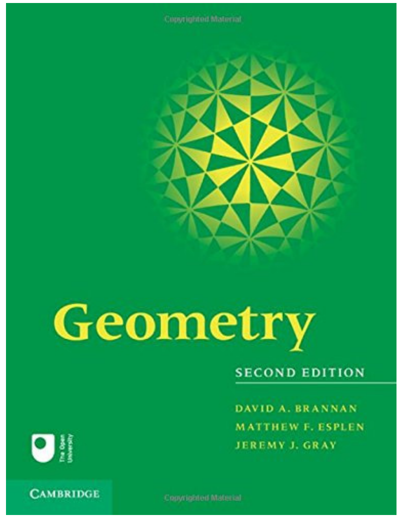
Unification

Conformal GA unifies Euclidean, projective, spherical, and hyperbolic geometries in a single compact framework.



Geometries and Klein

Understand geometries in terms of the underlying transformation groups



Euclidean

$$\boldsymbol{x} \mapsto \boldsymbol{U}\boldsymbol{x} + \boldsymbol{a}$$

Affine

$$\boldsymbol{x} \mapsto \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$$

Projective

$$[\boldsymbol{x}] \mapsto [\boldsymbol{A}\boldsymbol{x}]$$

Conformal

$$X \mapsto RX\tilde{R}$$

Mobius /Inversive

$$z \mapsto (az + b)/(cz + d)$$

Spherical

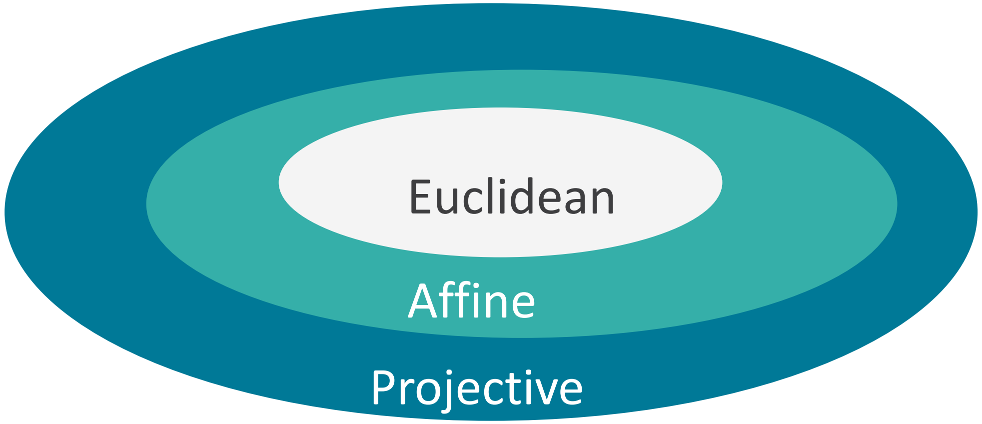
$$\hat{\boldsymbol{x}} \mapsto R\hat{\boldsymbol{x}}\tilde{R}$$

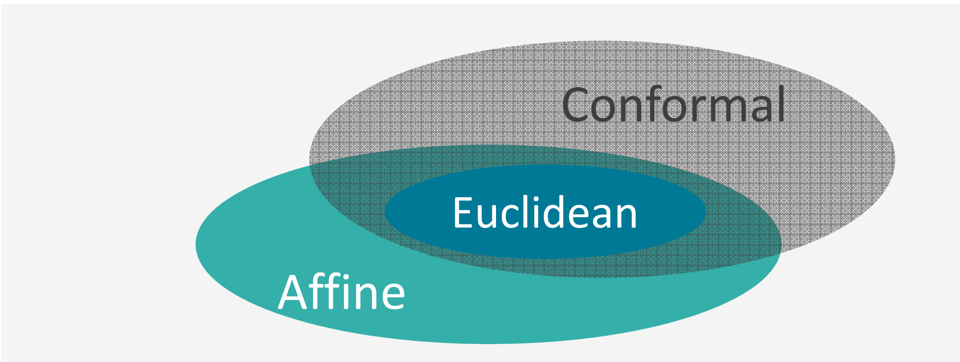
non-Euclidean

$$Re\tilde{R} = e$$

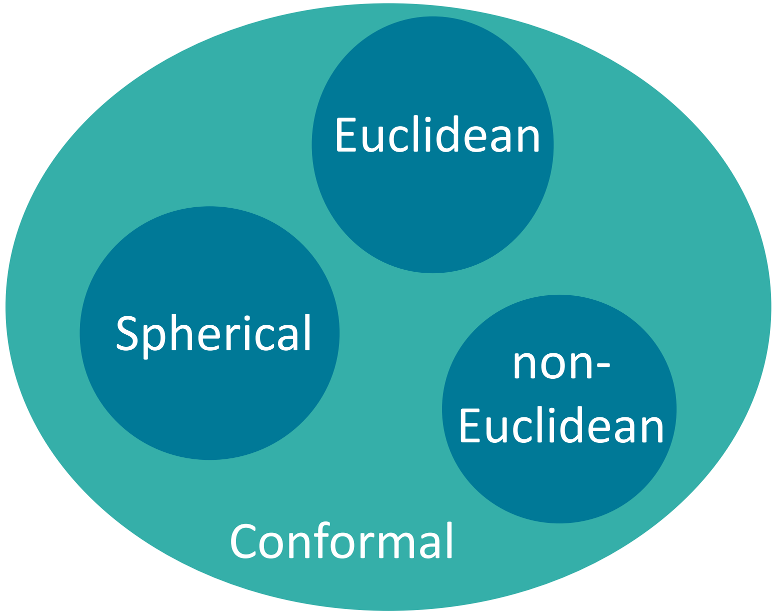
Geometries and Klein

Projective viewpoint





Conformal viewpoint



Groups

Have seen that we can perform dilations with rotors

Every linear transformation is rotation + dilation + rotation via SVD $A = U\Lambda V$

Trick is to double size of space

$$\{e_i, f_i\}, \quad e_i \cdot e_j = \delta_{ij}, \quad f_i \cdot f_j = -\delta_{ij}, \quad e_i \cdot f_j = 0$$

Null basis $n_i = e_i + f_i, \quad \bar{n}_i = e_i - f_i$

Define bivector

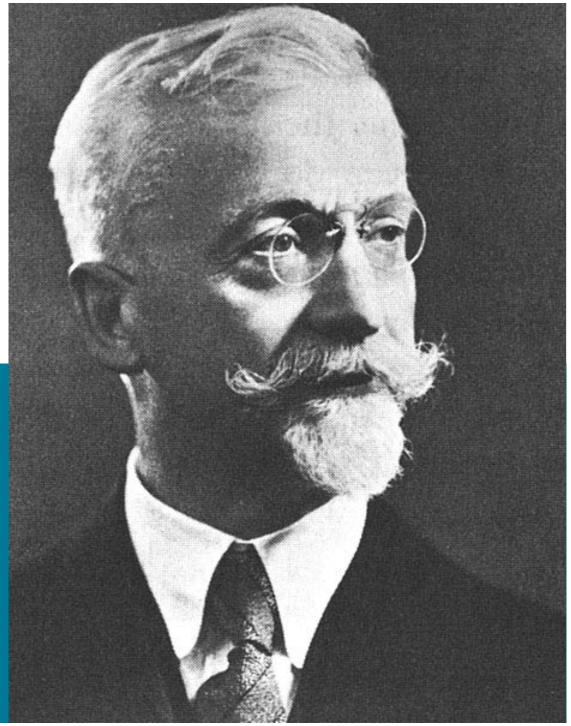
$$K = \sum_i e_i f_i \qquad (a \cdot K) \cdot K = a$$

Construct group from constraint

$$RK\tilde{R} = K$$

Keeps null spaces separate. Within null space give general linear group.

Unification

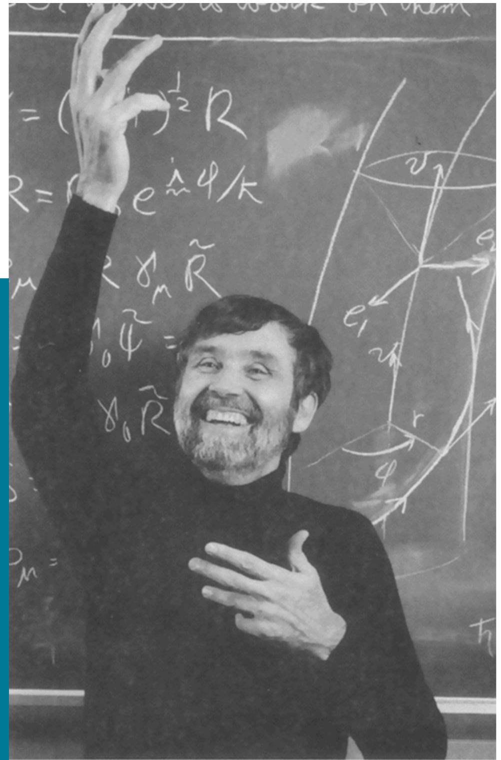


Every matrix group can be realised as a rotor group in some suitable space. There is often more than one way to do this.

Design of mathematics

Coordinate geometry
Complex analysis
Vector calculus
Tensor analysis
Matrix algebra
Lie groups
Lie algebras
Spinors
Gauge theory

Grassmann algebra
Differential forms
Berezin calculus
Twistors
Quaternions
Octonions
Pauli operators
Dirac theory
Gravity…



D Hestenes

Spinors and twistors

$$\sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\sigma_i \sigma_j = \delta_{ij} + i\epsilon_{ijk}\sigma_k$$

Spin matrices act on 2-component wavefunctions

These are spinors

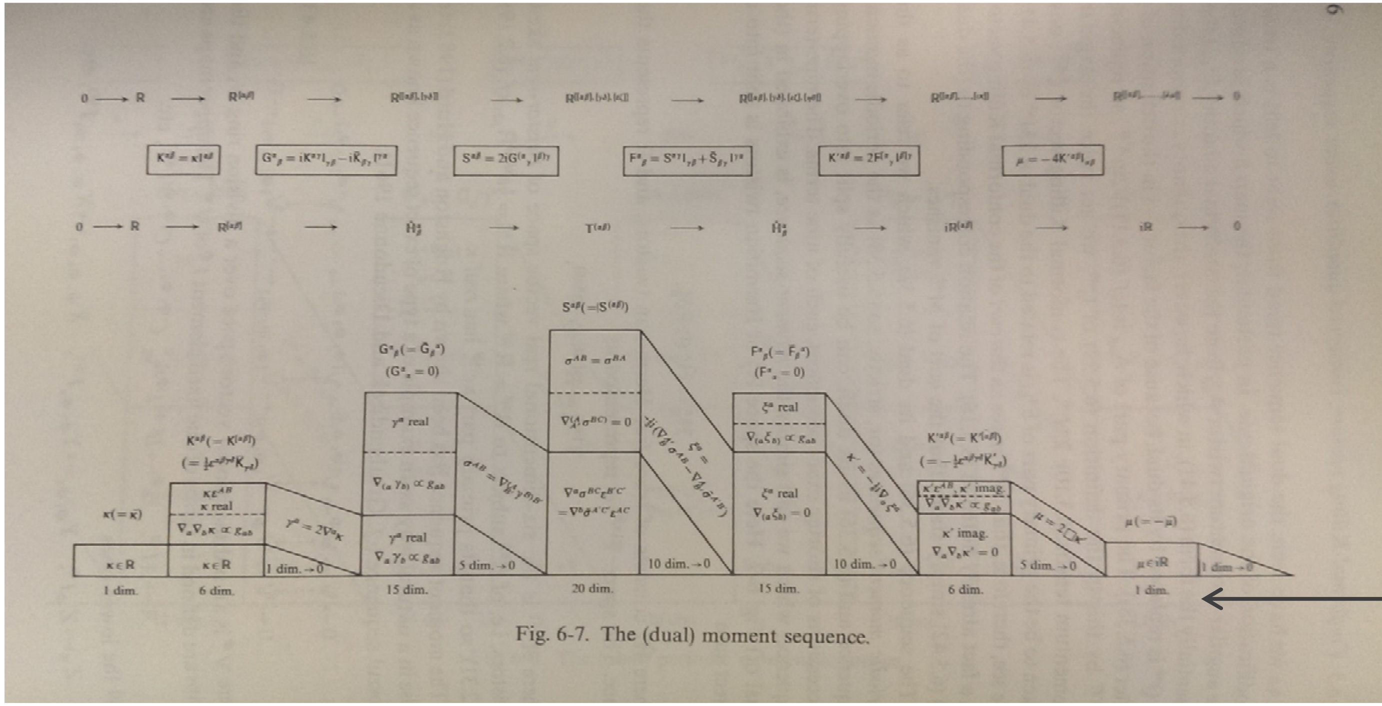
Very similar to qubits

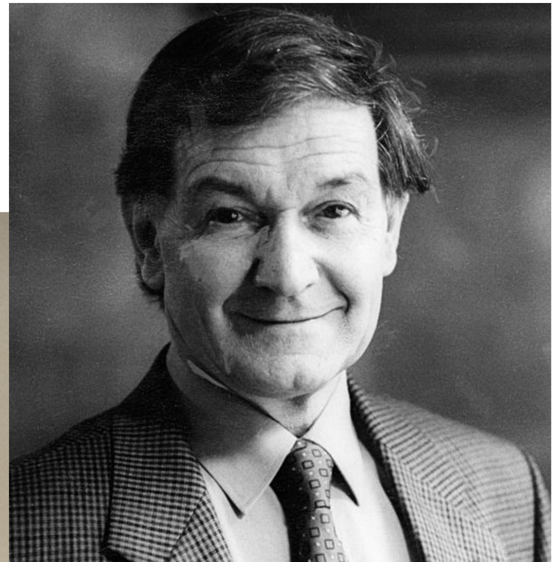
$$|\psi\rangle \mapsto \rho R$$

Roger Penrose has put forward a philosophy that spinors are more fundamental than spacetime

Start with 2-spinors and build everything up from there

Twistors





Look at dimensionality of objects in twistor space

Conformal GA of spacetime!

Forms and exterior calculus

Working with just the exterior product, exterior differential and duality recovers the language of forms

Motivation is that this is the ‘non-metric’ part of the geometric product

Interesting development to track is the subject of discrete exterior calculus

This has a discrete exterior product

$$\langle \alpha^k \wedge \beta^l, \sigma^{k+l} \rangle = \frac{1}{(k+l)!} \sum_{\tau \in S_{k+l+1}} \operatorname{sign}(\tau) \frac{|\sigma^{k+l} \cap \star v_{\tau(k)}|}{|\sigma^{k+l}|} \alpha \smile \beta(\tau(\sigma^{k+l})) ,$$

This is associative! Hard to prove.

Challenge – can you do better?

Implementation

1. What is the appropriate data structure?
2. How do we implement the geometric product
3. 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 compact data structure – hardware friendly • 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 • 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 and grade.

Linked list

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

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

As a linked list `(a1,1):(a2,2):(a3,8):[]`

For	Against
• Strongly typed • Sparse • 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 converting lists to arrays • Need a look-up table to store blade products

Linked list

Details depend on whether you want to use mixed signature space

Best to stay as general as possible

Blade	Binary	Integer
1	0	0
e1	1	1
f1	10	2
e2	100	4
f2	1000	8
e1f1	11	3
e1e2	101	5

Geometric product is an xorr operation

Implement this in a look-up table

Have to take care of sign

Careful with typographical ordering

Blade product

```
bladeprod (a,n) (b,m) = (x,r)
 where (fn,r) = bldprod n m
       x = fn (a*b)
```

The `bldprod` function must

1. Convert integers to binary 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

Geometric product

$$AB$$

`[Blades]` `[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

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 …
- Microsoft developing F#, and supporting Haskell

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

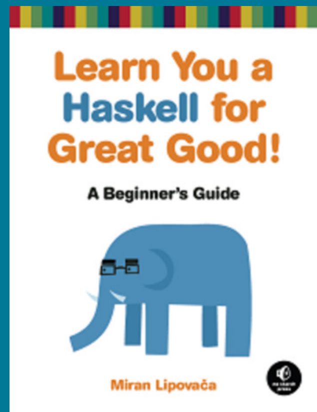
Haskell is strongly typed, and statically typed

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 …



learnyouahaskell.com

haskell.org/platform

wiki.haskell.org

Why Haskell?

Recursion

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

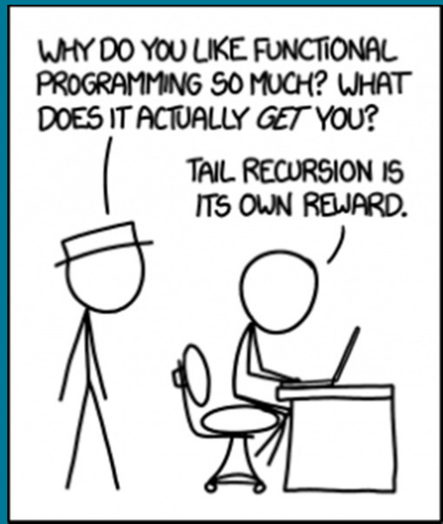
Laziness

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

$$\langle ABCD \rangle_0$$

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"



Resources

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