MULTI-ALGEBRA FLUENCY

C. Doran[a]

[a]Cambridge University
Cambridge, UK
cjld1@cam.ac.uk

Summary of the Abstract

When proving results 'by hand' in geometric algebra it is frequently helpful to move between different algebras, most commonly versions of Euclidean, projective and conformal algebras. This flexibility is harder to achieve in code without a sufficiently powerful type system. In this talk I describe a solution to the problem in the language of Julia, though many of the lessons hold for other languages. Julia [1] is dynamically typed and its type system contains two powerful features: parametric polymorphism and multiple dispatch. Utilising these we can develop code that can move between algebras with minimal overhead. We illustrate how this operates with a simple ray-tracing engine constructed entirely from GA primitives, with all source code available on GitHub.

Algebra Hopping

There is an unfortunate tendency in the discourse around geometric algebra for people to argue the merits of one particular instance of a geometric algebra versus another. This is very much against the spirit geometric calculus as described by David Hestenes and Garret Sobczyk [2]. There the authors imagine a universal algebra containing every finite algebra as a specific sub-case, defined by its own pseudoscalar. This is most elegantly employed in the theory of vector manifolds, where the pseudoscalar defining a manifold is pictured a living in some larger, unspecified embedding space.

Of course, a completely flexible picture such as this is much harder to achieve in code. Typically one wants to fix an algebra, so that all objects have a known size, and then define various properties of these objects. It is possible to work in a much more general way, by referencing basis blades as coming from an arbitrary-sized (n, n) balanced algebra, and then working out products as they are needed using a binary encoding trick. But such an approach is slow compared to a direct implementation of a specific algebra. There are two reasons for this. Each time a blade product is called we repeat a calculation to find the resulting basis blades, and working with lists of blades as the primary objects misses some crucial optimisations that are very helpful in the low dimensional algebras of most interest.

Having said all that, if you really wish to apply geometric algebra to problems in much higher dimensions, there is no option but to work with lists of blades as the primitive objects. If you tried to store a complete multivector in 60 dimensions you would quickly run out of memory! You have to move to some form of sparse representation.

The desire to write heavily optimised code for specific algebras does tend to tie people to choosing an algebra for a problem up front, and then sticking with it. This is generally ok for smaller problems. If you are working in 3D geometry, and expect to need rotors to carry out Euclidean transformations, but also have an idea that you might want to employ spheres somewhere in your problem, then you can simply opt to work in the conformal algebra of space, $G(4,1)$, and just live with the fact that this may be carrying round extra degrees of freedom that are unwanted. There will be a memory and performance hit, but that may be insignificant.
At the other extreme you could minimise memory and work in $G(3,0)$, and accept the fact that some of your algorithms are messy and lack the elegance (and often speed) provided by projective or conformal representations. Or you could take a half-way approach and work in the projective frameworks of $G(4,0)$ or $G(3,0,1)$, each with their own sets of benefits and trade-offs.
In practice, it is highly desirable to be able to move between algebras fluidly. This should be no surprise to anyone who has worked with graphics or computational geometry code. There one is frequently moving between 3D and 4D representations of objects, often with the 4th component set to one of 1 or 0 depending whether you are referring to a point or a vector. Our code would be simplified enormously if we were able to have multiple algebras in flight, with promotion rules between them that are fast, and logical and impose minimal overhead on the coder. Such a scheme is described here, with the accompanying code provided on GitHub [3].

The algebra relationships

One way to understand how to jump between algebras is through the defining relationships between them, and how these define the underlying representations of the algebras over reals, complex numbers or quaternions.
There are three relationships for jumping up or down 2 dimensions:

$$\begin{aligned} G(p+1,q+1) &= G(p,q) \otimes G(1,1) \\ G(q+2,p) &= G(p,q) \otimes G(2,0) \\ G(q,p+2) &= G(p,q) \otimes G(0,2) \end{aligned} \tag{1}$$

Of these the first is by far the most significant, as it defnines the conformal embedding of an algebra inside a larger one. This allows us to write the elements of $G(p+1,q+1)$ as

$$\{1, e_1, f_1, e_1 f_1\} \otimes G(p,q) \tag{2}$$

In this decomposition all generators on the left commute with all generators on the right. This is achieved by making the 'vectors' in $G(p,q)$ out of trivectors in $G(p+1,q+1)$. This does make sense as the generators of $G(p,q)$ are then lines in $G(p+1,q+1)$ through a common origin. But this embedding may not be precisely the one we want. Typically we want to embed points in, say, $G(3,0)$ as null vectors in $G(4,1)$. Our scheme does need to flexible enough to cover all of these possibilities.
The second relationship between algebras is the one governing the even sub-algebra (ESA). We have

$$G(p,q)^+ = G(q,p-1) = G(p,q-1) \tag{3}$$

This is helpful because we can save space in our implementation of an algebra by just implementing the ESA, together with a rule for moving between even and odd elements. To see how this works, consider the simple case of $G(3,0)$. As a complete algebra this can be realised as 2×2 complex matrices (the Pauli matrices). But in practice we never want to combine even and odd grade objects. On the rare occasions where there is a strong reason to combine even and odd elements this is telling us we should be working in a higher-dimensional algebra. For example, it does make sense to combine the electric field (E, a vector in $G(3,0)$) and the magnetic field (B, a bivector in $G(3,0)$). But the correct way to do this is in the spacetime algebra, where the 6 bivector degrees of freedom are just what we need to combine E and B.
If we only want to store even elements for $G(3,0)$ we need to look at the even sub-algebra

$$G(3,0)^+ = H. \tag{4}$$

This is simply the quaternion algebra. So we need a fast implementation of the quaternion algebra to perform multiplications of even-grade elements. We also need a map from odd elements to even. This is straightforward as in all odd-dimensional algebras the map is provided by the pseudoscalar:

$$O \mapsto -e_1 e_2 e_3 O = E \tag{5}$$

where O and E are the odd and even-grade objects respectively. The objects we store are quaternions, which are simply a `struct` with 4 entries. These are compact, and relate naturally to the objects we expect to use in 3D and 4D representations of geometry. So, for example, a vector in 3D is stored as a `struct` with entries $(x, y, z, 0)$, where x, y and z are coordinates [1].
The remaining algebras that most frequently arise in applications to 3D geometry are the two 4-dimensional algebras $G(4,0)$ and $G(3,0,1)$. The first of these is the appropriate framework for projective geometry, and its ESA decomposes neatly into

$$G(4,0)^+ = H \oplus H \tag{6}$$

An even element in this algebra consists of a pair of quaternions. We can immediately lift a 3D vector, represented as the quaternion q, into $G(4,0)$ by defining

$$q \mapsto (q, -q). \tag{7}$$

And this operation can be performed on the fly. If the compiler sees we are trying to multiply a 3D object with a 4D object, this map can be performed and the result compute in $G(4,0)$.
The second 4-dimensional algebra of interest is $G(3,0,1)$, the algebra associated with the Euclidean group of rotations and translations, and where the grade-1 objects are planes. In this case we also find

$$G(3,0,1)^+ = H \oplus H \tag{8}$$

[1] I should apologise to the geometric algebra gods for having to resort to coordinates (and matrices) so soon. But to write performant code you have to make choices about what and how you are storing. All code can still be written 'coordinate-free'.

and again the ESA is represented by a quaternion pair. In this case the lifting operation requires a but more care. But assuming we want to lift a point in 3D (again, the quaternion q) up to the equivalent trivector in $G(3,0,1)$ then the operation is simply

$$q \mapsto (0, q). \tag{9}$$

Again, there is no need to modify the underlying quaternion.
Finally, the last algebra of direct relevance is the conformal algebra $G(4,1)$. The ESA of this is represented by

$$G(4,1)^+ = H(2) \tag{10}$$

so this time we form a 2×2 matrix of quaternions. It is remarkable that in all cases we are led back to quaternions as the primitive object in very much the same way the Clifford found that the quaternions were central to his original research. For the conformal case the lift operation is still simple

$$q \mapsto \begin{pmatrix} q & 0 \\ 0 & -q \end{pmatrix} \tag{11}$$

1 The type system

This section describes the type system used in Julia to distinguish between the algebras, and to allow rapid interchange between them. This is being worked on currently. We define two functions, `point` and `vector`, for lifting objects from 3D. The former puts a 1 into the 4th component, and the latter does not. The former also ensures that the lifted point is a null vector in the conformal representation. Naming conventions and other details are being worked on.

Acknowledgements

I am grateful to everyone at Monumo for their continued support. I am particularly grateful to Tom Gilbert for all his work on the SimpleGA codebase, turning it from a personal hobby to something that should be useful to anyone looking to explore geometric algebra in Julia.

References

[1] Bezanson, Jeff and Karpinski, Stefan and Shah, Viral B and Edelman, Alan, Julia: A fast dynamic language for technical computing. *arXiv:1209.5145 2010.*, 2012

[2] Hestenes, D. and Sobczyk, G.,Clifford Algebra to Geometric Calculus, Reidel, Dordrecht, 1984

[3] SimpleGA, github.com/MonumoLtd/SimpleGA.jl