1 WHY GEOMETRIC ALGEBRA?

This book is about geometric algebra, a powerful computational system to describe and solve geometrical problems. You will see that it covers familiar ground—lines, planes, spheres, rotations, linear transformations, and more—but in an unfamiliar way. Our intention is to show you how basic operations on basic geometrical objects can be done differently, and better, using this new framework.

The intention of this first chapter is to give you a fair impression of what geometric algebra can do, how it does it, what old habits you will need to extend or replace, and what you will gain in the process.

1.1 AN EXAMPLE IN GEOMETRIC ALGEBRA

To convey the compactness of expression of geometric algebra, we give a brief example of a geometric situation, its description in geometric algebra, and the accompanying code that executes this description. It helps us discuss some of the important properties of the computational framework. You should of course read between the lines: you will be able to understand this example fully only at the end of Part II, but the principles should be clear enough now.

Suppose that we have three points c_1, c_2, c_3 in a 3-D space with a Euclidean metric, a line L, and a plane Π. We would like to construct a circle C through the three points, rotate it around the line L, and then reflect the whole scene in the plane Π. This is depicted in Figure 1.1. Here is how geometric algebra encodes this in its conformal model of Euclidean geometry:

1. ***Circle***. The three points are denoted by three elements c_1, c_2, and c_3. The oriented circle through them is
$$C = c_1 \wedge c_2 \wedge c_3.$$
The $\wedge$ symbol denotes the *outer product*, which constructs new elements of computation by an algebraic operation that geometrically connects basic elements (in this case, it connects points to form a circle). The outer product is antisymmetric: if you wanted a circle with opposite orientation through these points, it would be $-C$, which could be made as $-C = c_1 \wedge c_3 \wedge c_2$.
2. ***Rotation***. The rotation of the circle C is made by a sandwiching product with an element R called a rotor, as
$$C \mapsto R\,C/R.$$

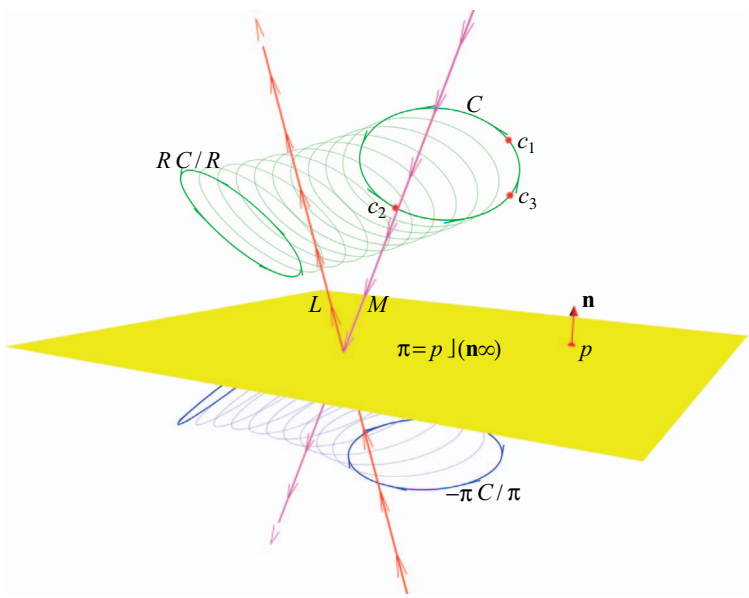

Figure 1.1: The rotation of a circle C (determined by three points c_1, c_2, c_3) around a line L, and the reflections of those elements in a plane Π.

The product involved here is the *geometric product*, which is the fundamental product of geometric algebra, and its corresponding division. The geometric product multiplies transformations. It is structure-preserving, because the rotated circle through three points is the circle through the three rotated points:

$$R\,(c_1 \wedge c_2 \wedge c_3)/R = (Rc_1/R) \wedge (R\,c_2/R) \wedge (R\,c_3/R).$$

Moreover, any element, not just a circle, is rotated by the same rotor-based formula. We define the value of the rotor that turns around the line L below.

3. ***Line***. An oriented line L is also an element of geometric algebra. It can be constructed as a "circle" passing through two given points a_1 and a_2 and the point at infinity ∞, using the same outer product as in item 1:

$$L = a_1 \wedge a_2 \wedge \infty.$$

Alternatively, if you have a point on L and a direction vector $\mathbf{u}$ for L, you can make the same element as

$$L = a_1 \wedge \mathbf{u} \wedge \infty.$$

This specifies exactly the same element L by the same outer product, even though it takes different arguments. This algebraic equivalence saves the construction of many specific data types and their corresponding methods for what are geometrically the same elements.

The point at infinity ∞ is an essential element of this operational model of Euclidean geometry. It is a finite element of the algebra, with well-defined algebraic properties.

4. ***Line Rotation***. The rotor that represents a rotation around the line L, with rotation angle ϕ, is

$$R = \exp(\phi\, L^*/2).$$

This shows that geometric algebra contains an exponentiation that can make elements into rotation operators. The element L^* is the *dual* of the line L. Dualization is an operation that takes the geometric complement. For the line L, its dual can be visualized as the nest of cylinders surrounding it.

If you would like to perform the rotation in N small steps, you can interpolate the rotor, using its logarithm to compute $R^{1/N}$, and applying that n times (we have done so in Figure 1.1, to give a better impression of the transformation). Other transformations, such as general rigid body motions, have logarithms as well in geometric algebra and can therefore be interpolated.

5. ***Plane***. To reflect the whole situation with the line and the circles in a plane Π, we first need to represent that plane. Again, there are alternatives. The most straightforward is to construct the plane with the outer product of three points p_1, p_2, p_3 on the plane

and the point at infinity ∞, as $\Pi = p_1 \wedge p_2 \wedge p_3 \wedge \infty$. Alternatively, we can instead employ a specification by a normal vector $\mathbf{n}$ and a point p on the plane. This is a specification of the dual plane $\pi \equiv \Pi^*$, its geometric complement:

$$\pi = p \rfloor (\mathbf{n}\infty) = \mathbf{n} - (\mathbf{p} \cdot \mathbf{n})\,\infty.$$

Here $\rfloor$ is a contraction product, used for metric computations in geometric algebra; it is a generalization of the inner product (or dot product) from vectors to the general elements of the algebra. The duality operation above is a special case of the contraction.

The change from p to $\mathbf{p}$ in the equation is not a typo: p denotes a point, $\mathbf{p}$ is its location vector relative to the (arbitrary) origin. The two entities are clearly distinct elements of geometric algebra, though computationally related.

6. ***Reflection***. Either the plane Π or its geometric complement π determine a reflection operator. Points, circles, or lines (in fact, any element X) reflect in the plane in the same way:

$$X \mapsto -\pi X/\pi.$$

Here the reflection plane π, which is an oriented object of geometric algebra, acts as a reflector, again by means of a sandwiching using the geometric product. Note that the reflected circle has the proper orientation in Figure 1.1.

As with the rotation in item 2, there is obvious structure preservation: the reflection of the rotated circle is the rotation of the reflected circle (in the reflected line). We can even reflect the rotor to become $R' \equiv \pi \exp(\phi\, L^*/2)/\pi = \exp(-\phi\,(-\pi L^*/\pi)/2)$, which is the rotor around the reflected line, automatically turning in the opposite orientation.

7. ***Programming***. In total, the scene of Figure 1.1 can be generated by a simple C++ program computing directly with the geometric objects in the problem statement, shown in Figure 1.2. The outcome is plotted immediately through the calls to the multivector drawing function `draw()`. And since it has been fully specified in terms of geometric entities, one can easily change any of them and update the picture. The computations are fast enough to do this and much more involved calculations in real time; the rendering is typically the slowest component.

Although the language is still unfamiliar, we hope you can see that this is geometric programming at a very desirable level, in terms of quantities that have a direct geometrical meaning. Each item occurring in any of the computations can be visualized. None of the operations on the elements needed to be specified in terms of their coordinates. Coordinates are only needed when entering the data, to specify precisely which points and lines are to be operated upon. The absence of this quantitative information may suggest that geometric algebra is merely an abstract specification language with obscure operators that merely convey the mathematical logic of geometry. It is much more than that: all expressions are quantitative prescriptions of computations, and can be executed directly. Geometric algebra is a programming language, especially tailored to handle geometry.

```
// l1, l2, c1, c2, c3, p1 are points
// OpenGL commands to set color are not shown
line L; circle C; dualPlane p;

L = unit_r(l1 ^ l2 ^ ni);
C = c1 ^ c2 ^ c3;
p = p1 << (e2 ^ ni);

draw(L); // draw line (red)
draw(C); // draw cicle (green)
draw(p); // draw plane (yellow)

draw(-p * L * inverse(p)); // draw reflected line (magenta)
draw(-p * C * inverse(p)); // draw reflected circle (blue)

// compute rotation versor:
const float phi = (float)(M_PI / 2.0);
TRversor R;
R = exp(0.5f * phi * dual(L));

draw(R * C * inverse(R)); // draw rotated cicle (green)

// draw reflected, rotated circle (blue)
draw(-p * R * C * inverse(R) * inverse(p));

// draw interpolated circles
pointPair LR = log(R); // get log of R
for (float alpha = 0; alpha < 1.0; alpha += 0.1f)
{
  // compute interpolated rotor
  TRversor iR;
  iR = exp(alpha * LR);

  // draw rotated circle (light green)
  draw(iR * C * inverse(iR));

  // draw reflected, rotated circle (light blue)
  draw(-p * iR * C * inverse(iR) * inverse(p));
}
```

Figure 1.2: Code to generate Figure 1.1.

You may be concerned about the many different products that occurred in this application. If geometric algebra needs a new product for every new operation, its power would be understandable, but the system would rapidly grow unwieldy. This is perhaps the biggest surprise of all: *there is only one product that does it all*. It is the *geometric product*

(discovered by William Kingdon Clifford in the 1870s), which we used implicitly in the example in the sandwiching operations of rotation and reflection. The other products ($\wedge$, $\rfloor$, *, sandwiching) are all specially derived products for the purposes of spanning, metric projection, complementation, and operating on other elements. They can all be defined in terms of the geometric product, and they correspond closely to how we think about geometry classically. That is the main reason that they have been given special symbols. Once you get used to them, you will appreciate the extra readability they offer. But it is important to realize that you really only need to implement one product to get the whole consistent functionality of geometric algebra.

Because of the structural properties of geometric algebra, this example can be extended in many ways. To name a few:

- ***Spherical Reflection***. If we had instead wanted to reflect this situation in a sphere, this is done by

$$X \mapsto -\sigma X/\sigma.$$

  Here σ is the dual representation of a sphere (it encodes a sphere with center c passing through p as the representational vector $p \rfloor (c \wedge \infty)$). We depict this in Figure 1.3.

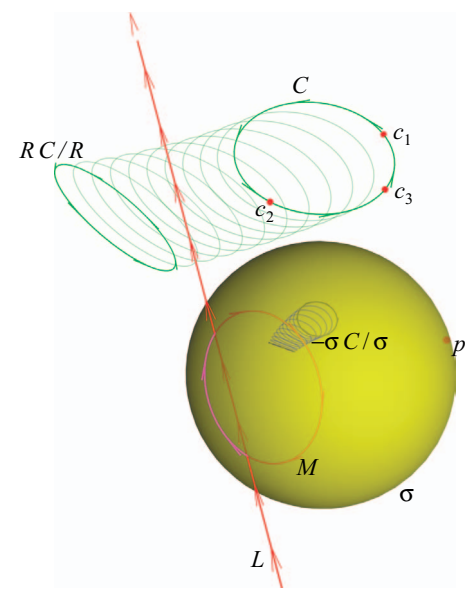

Figure 1.3: The rotation of a circle C (determined by three points c_1, c_2, c_3) around a line L, and the reflections of those elements in a sphere σ.

The only thing that is different from the program generating Figure 1.1 is that the plane π was replaced by the sphere σ, not only geometrically, but also algebraically. This generates the new reflection, which reflects the line L to become the circle $M = -\sigma L/\sigma$. It also converts the reflected rotor around M into the operation $\sigma R/\sigma$, which generates a scaled rotation around a circle, depicted in the figure. The whole structure of geometric relationships is nicely preserved.

- ***Intersections***. The π-based reflection operator of item 6 takes the line L and produces the reflected line $\pi L/\pi$, without even computing the intersection point of the line and the plane. If we had wanted to compute the intersection of line and plane, that would have been the point $\pi \rfloor L = \Pi^* \rfloor L$. This is another universal product, the `meet`, which computes the intersection of two elements Π and L.
- ***Differentiation***. It is even possible to symbolically differentiate the final expression of the reflected rotated circle to any of the geometrical elements occurring in it. This permits a sensitivity analysis or a local linearization; for instance, discovering how the resulting reflected rotated circle would change if the plane π were to be moved and tilted slightly.

1.2 HOW IT WORKS AND HOW IT'S DIFFERENT

The example has given you an impression of what geometric algebra can do. To understand the structure of the book, you need a better feeling for what geometric algebra is, and how it relates to more classical techniques such as linear algebra.

The main features of geometric algebra are:

- ***Vector Spaces as Modeling Tools***. Vectors can be used to represent aspects of geometry, but the precise correspondence is a modeling choice. Geometric algebra offers three increasingly powerful models for Euclidean geometry.
- ***Subspaces as Elements of Computation***. Geometric algebra has products to combine vectors to new elements of computation. They represent oriented subspaces of any dimension, and they have rich geometric interpretations within the models.
- ***Linear Transformations Extended***. A linear transformation on the vector space dictates how subspaces transform; this augments the power of linear algebra in a structural manner to the extended elements.
- ***Universal Orthogonal Transformations***. Geometric algebra has a special representation of orthogonal transformations that is efficient and universally applicable in the same form to all geometric elements.
- ***Objects are Operators***. Geometric objects and operators are represented on a par, and exchangeable: objects can act as operators, and operators can be transformed like geometrical objects.

- ***Closed Form Interpolation and Perturbation***. There is a geometric calculus that can be applied directly to geometrical objects and operators. It allows straightforward interpolation of Euclidean motions.

In the following subsections, we elaborate on each of these topics.

1.2.1 VECTOR SPACES AS MODELING TOOLS

When you use linear algebra to describe the geometry of elements in space, you use a real vector space $\mathbb{R}^m$. Geometric algebra starts with the same domain. In both frameworks, the vectors in an m-dimensional vector space $\mathbb{R}^m$ represent 1-D directions in that space. You can think of them as denoting lines through the origin. To do geometry flexibly, we want more than directions; we also want *points* in space. The vector space $\mathbb{R}^m$ does not have those by itself, though its vectors can be used to represent them.

Here it is necessary to be more precise. There are two structures involved in doing geometrical computations, both confusingly called "space."

- There is the physical 3-D space of everyday experience (what roboticists call the task space). It contains the objects that we want to describe computationally, to move around, to analyze data about, or to simply draw.
- Mathematics has developed the concept of a vector space, which is a space of abstract entities with properties originally inspired by the geometry of physical space.

Although an m-dimensional vector space is a mathematical generalization of 3-D physical space, it does not follow that 3-D physical space is best described by a 3-D vector space. In fact, in applications we are less interested in the *space* than in the *geometry*, which concerns the objects residing in the space. That geometry is defined by the motions that can freely move objects. In Euclidean geometry, those motions are translations, rotations, and reflections. Whenever two objects differ only by such transformations we refer to them as the same object, but at a different location, with a different orientation, or viewed in a mirror. (Sometimes *scaling* is also included in the permitted equivalences.)

So we should wonder what computational model, based in a vector space framework, can conveniently represent these natural motions of Euclidean geometry. Since the motions involve certain measures to be preserved (such as size), we typically use a metric vector space to model it. We present three possibilities that will recur in this book:

1. ***The Vector Space Model***. A 3-D vector space with a Euclidean metric is well suited to describe the algebra of *directions* in 3-D physical space, and the operation of rotation that transforms directions. Rotations (and reflections) are orthogonal linear transformations: they preserve distances and angles. They can be represented by 3×3 orthogonal matrices or as quaternions (although the latter are not in the *linear* algebra of $\mathbb{R}^3$, we will see that they are in the *geometric* algebra of $\mathbb{R}^3$).
2. ***The Homogeneous Model***. If you also want to describe translations in 3-D space, it is advantageous to use *homogeneous coordinates*. This employs the *vectors* of a

4-D vector space to represent *points* in physical 3-D space. Translations now also become linear transformations, and therefore combine well with the 3-D matrix representation of rotations.

The extra fourth dimension of the vector space can be interpreted as the *point at the origin* in the physical space. There is some freedom in choosing the metric of this 4-D vector space, which makes this model suitable for projective geometry.

3. ***The Conformal Model***. If we want the translations in 3-D physical space represented as *orthogonal* transformations (just as rotations were in the 3-D vector space model), we can do so by employing a 5-D vector space. This 5-D space needs to be given a special metric to embed the metric properties of Euclidean space. It is expressed as $\mathbb{R}^{4,1}$, a 5-D vector space with a Minkowski metric.

   The vectors of the vector space $\mathbb{R}^{4,1}$ can be interpreted as dual spheres in 3-D physical space, including the zero-radius spheres that are points. The two extra dimensions are the *point at the origin* and the *point at infinity*.

   This model was used in the example of Figure 1.1. It is called the conformal model because we get more geometry than merely the Euclidean motions: all conformal (i.e., angle-preserving) transformations can be represented as orthogonal transformations. One of those is inversion in a sphere, which explains why we could use a spherical reflector in Figure 1.3.

Although these models can all be treated and programmed using standard linear algebra, there is great advantage to using geometric algebra instead:

- Geometric algebra uses the subspace structure of the vector spaces to construct extended objects.
- Geometric algebra contains a particularly powerful method to represent orthogonal transformations.

The former is useful to all three models of Euclidean geometry; the latter specifically works for the first and third. In fact, the conformal model was invented before geometric algebra, but it lay dormant. Only with the tools that geometric algebra offers can we realize its computational potential. We will treat all these models in Part II of this book, with special attention to the conformal model. In Part I, we develop the techniques of geometric algebra, and prefer to illustrate those with the more familiar vector space model, to develop your intuition for its computational capabilities.

1.2.2 SUBSPACES AS ELEMENTS OF COMPUTATION

Whatever model you use to describe the geometry of physical space, understanding vector spaces and their transformations is a fundamental prerequisite. Linear algebra gives you techniques to compute with the basic elements (the vectors) by using matrices. Geometric algebra focuses on the *subspaces of a vector space as elements of computation*. It constructs these systematically from the underlying vector space, and extends the matrix techniques

to transform them, even supplanting those completely when the transformations are orthogonal.

The outer product $\wedge$ has the constructive role of making subspaces out of vectors. It uses k independent vectors $\mathbf{v}_i$ to construct the computational element $\mathbf{v}_1 \wedge \mathbf{v}_2 \wedge \cdots \wedge \mathbf{v}_k$, which represents the k-dimensional subspace spanned by the $\mathbf{v}_i$. Such a subspace is *proper* (also known as *homogeneous*): it contains the origin of the vector space, the zero vector 0. An m-dimensional vector space has many independent proper subspaces: there are $\binom{m}{k}$ subspaces of k dimensions, for a total of 2^m subspaces of any dimension. This is a considerable amount of structure that comes for free with the vector space $\mathbb{R}^m$, which can be exploited to encode geometric entities.

Depending on how the vector space $\mathbb{R}^m$ is used to model geometry, we obtain different geometric interpretations of its outer product.

- In the vector space model, a vector represents a 1-D direction in space, which can be used to encode the direction of a line through the origin. This is a 1-D proper subspace. The outer product of two vectors then denotes a 2-D direction, which signifies the attitude of an oriented plane through the origin, a 2-D proper subspace of the vector space. The outer product of three vectors is a volume. Each of those has a magnitude and an orientation. This is illustrated in Figure 1.4(a,b).
- In the homogeneous model, a *vector* of the vector space represents a *point* in the physical space it models. Now the outer product of two vectors represents an oriented line in the physical space, and the outer product of three vectors is interpreted as an oriented plane. This is illustrated in Figure 1.4(c,d). By the way, this representation of lines is the geometric algebra form of Plücker coordinates, now naturally embedded in the rest of the framework.
- In the conformal model, the points of physical space are viewed as spheres of radius zero and represented as vectors of the vector space. The outer product of three points then represents an oriented circle, and the outer product of four points an oriented sphere. This is illustrated in Figure 1.4(e,f). If we include the point at infinity in the outer product, we get the "flat" elements that we could already represent in the homogeneous model, as the example in Section 1.1 showed.

It is very satisfying that there is one abstract product underlying such diverse constructions. However, these varied geometrical interpretations can confuse the study of its algebraic properties, so when we treat the outer product in Chapter 2 and the rest of Part I, we prefer to focus on the vector space model to guide your intuitive understanding of geometric algebra. In that form, the outer product dates back to Hermann Grassmann (1840) and is the foundation of the *Grassmann algebra* of the extended quantities we call proper subspaces. Grassmann algebra is the foundation of geometric algebra.

In standard linear algebra, subspaces are not this explicitly represented or constructed. One can assemble vectors $\mathbf{v}_i$ as columns in a matrix $[\![\mathbb{V}]\!] = [\![\mathbf{v}_1\ \mathbf{v}_2 \cdots \mathbf{v}_k]\!]$, and then treat the image of this matrix, $\mathrm{im}([\![\mathbb{V}]\!])$, as a representation of the subspace, but this is not an

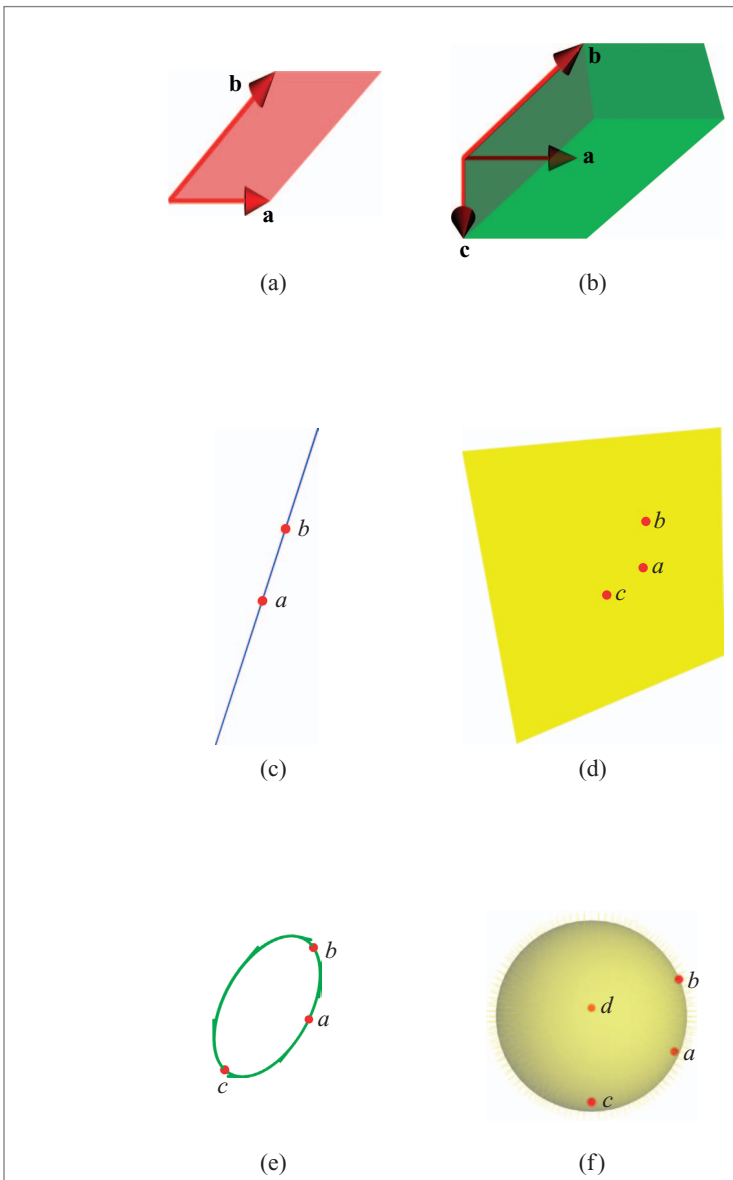

Figure 1.4: The outer product and its interpretations in the various models of Euclidean geometry. (a,b): the vector space model; (c,d): the homogeneous model; and (e,f): the conformal model.

integral part of the algebra; it is not a product in the same sense that the dot product is. If the matrix is square, we can take the determinant $\det([\![\mathbb{V}]\!])$ to represent the amount of area or volume of the subspace and its orientation, but if it is not square, such measures are less easily represented. Subspaces are simply not well represented in standard linear algebra.

1.2.3 LINEAR TRANSFORMATIONS EXTENDED

Linear transformations are defined by how they transform vectors in the vector space $\mathbb{R}^m$. As these vectors transform, so do the subspaces spanned by them. That fully defines how to extend a linear transformation to the subspace structure.

If one uses a matrix for the representation of the linear transformation on the vector space level, it is straightforward and automatic to extend this to a matrix that works on the subspace levels. You just take the outer product of its action on the basis vectors as its definition on the basis for subspaces. Now you can perform the same linear transformation on any subspace.

This way of thinking about linear transformations, with its use of the outer product, already provides structural advantages over the usual coordinate-based methods. Programs embedding this automatic transference of a vector space mapping to its subspaces are simpler. Moreover, they permit one to choose a representation for geometric elements that transforms most simply within this framework. An example is the representation of the attitude of a plane through the origin; its representation by a normal vector has more complicated transformations than its equally valid representation by an outer product of two vectors.

Within the subspace representation, a general product can be given for the intersection of subspaces (the `meet` product), which also transform in a structure-preserving manner under the extended linear transformations (the transform of an intersection is the intersection of the transforms). This uses more than the outer product alone; it also requires the contraction, or dualization.

The resulting consistent *subspace algebra* is good to understand first. Its subspace products are the algebraic extensions of familiar techniques in standard linear algebra. Seeing them in this more general framework will improve the way you program in linear algebra, even if you do not make explicit use of the extended data structures that the subspaces provide. Therefore we begin our journey with the treatment of this subspace algebra, in Chapters 2 to 5.

1.2.4 UNIVERSAL ORTHOGONAL TRANSFORMATIONS

In the vector space model and the conformal model, *orthogonal transformations* are used to represent basic motions of Euclidean geometry. This makes that type of linear transformation fundamental to doing geometry in those models. (General linear transformations are still useful to represent deformations of elements, on top of the basic motions of the geometry, but they are not as crucial).

Geometric algebra has a special way to represent orthogonal transformations, more powerful than using orthogonal matrices. These are *versors*, and the example in Section 1.1 showed two instances of them: a rotor and a reflector. A versor V transforms any element X of the geometric algebra according to the versor product:

$$X \mapsto (-1)^{xv} V X / V,$$

where the sign factor depends on the dimensionality of X and V, and need not concern us in this introduction. This operator product transcends matrices in that it can act directly on arbitrary elements: vectors, subspaces, and operators.

The product involved in the sandwiching of the versor product is the geometric product; as a consequence, subsequent operators multiply by the geometric product. For instance, $R_2\,(R_1\,X/R_1)/R_2 = (R_2\,R_1)\,X/(R_2\,R_1)$. This product is linear, associative, and invertible, but not commutative. That matches its geometric interpretation: orthogonal transformations are linear, associative, and invertible, but their order matters.

The two-sidedness of the versor product of an operator may come as a bit of a surprise, but you probably have seen such two-sided products before in a geometrical context.

- When the vectors of a space transform by a motion represented by $[\![\mathtt{M}]\!]$ (so that $[\![\mathbf{x}]\!]$ becomes $[\![\mathtt{M}]\!]\,[\![\mathbf{x}]\!]$), a matrix $[\![\mathtt{A}]\!]$ transforms to become $[\![\mathtt{M}]\!]\,[\![\mathtt{A}]\!]\,[\![\mathtt{M}]\!]^{-1}$. Note that in linear algebra, vectors and operators transform differently, whereas in geometric algebra they transform in the same manner.
- Another classical occurrence of the two-sided product is the quaternion representation of 3-D rotations. Those are in fact rotors and, therefore, versors. In the classical representation, you need to employ three imaginary numbers to represent them properly. We will see in Chapter 7 how geometric algebra simply uses the real subspaces of a 3-D vector space to construct quaternions. Quaternions are not intrinsically imaginary! Moreover, when given this context, they become universal operators, capable of rotating geometric subspaces (rather than only being applicable to other quaternions).

The versor form of an orthogonal transformation automatically guarantees the preservation of algebraic structure (more technically known as covariance). Geometrically, this implies that *the construction of an object from moved components equals the movement of the object constructed from the original components*. Here, "construction" can be the connection of the outer product, the intersection of the `meet`, the complementation of the duality operation, or any other geometrically significant operation.

You have seen in the example how this simplifies constructions. In traditional linear algebra, one can only transform *vectors* properly, using the matrices. So to move any construction one has built, one has to move the vectors on which it was based and rebuild it from scratch. With geometric algebra, it is possible to move the construction itself: the lines, circles, and other components, and moreover *all* of these are moved by the same versor construction with the same versor representing the motion.

You need no longer to be concerned about the type of element you are moving; they all automatically transform correctly. That means you also do not need to invent and build special functions to move lines, planes, or normal vectors and can avoid defining a motion method for each data structure, because all are generic. In fact, those differing methods may have been one of the reasons that forced you to distinguish the types in the first place. Now even that is not necessary, because they all find their place algebraically rather than by explicit construction, so fewer data types are required. This in turn reduces the number of cases in your program flow, and therefore may ultimately simplify the program itself.

1.2.5 OBJECTS ARE OPERATORS

In geometric algebra, operators can be specified directly in terms of geometric elements intrinsic to the problem.

We saw in Section 1.1, item 6, how the dual plane π (i.e., an object) could be used immediately as the reflector (i.e., an operator) to produce the reflected line and circles. We also constructed the rotor representing the rotation around the line L by exponentiating the line in item 4.

Geometric algebra offers a range of constructions to make versors. It is particularly simple to make the versors representing basic motions as ratios (i.e., using the division of the geometric product): the ratio of two planes is a rotation versor, the ratio of two points is a translation versor, and the ratio of two lines in 3-D is the screw motion that turns and slides one into the other. These constructions are very intuitive and satisfyingly general.

As you know, it is much harder to define operators in such direct geometrical terms using linear algebra. We summarize the usual techniques:

- There are several methods to construct rotation operators. Particularly intricate are various kinds of standardized systems of orientating frames by subsequent rotations around Euler angles, a source of errors due to the arbitrariness of the coordinate frames. One can construct a rotation matrix from the rotation axis directly (by Rodrigues' formula), and this is especially simple for a quaternion (which is already an element of geometric algebra). Unfortunately, even those are merely rotations at the origin. There is no simple formula like the $\exp(\phi L^*/2)$ of geometric algebra to convert a general axis L into a rotation operator.
- Translations are defined by the difference of vectors, which is simple enough, but note that it is a different procedure from defining rotations.
- A general rigid body motion converting one frame into another can be artificially split in its rotational aspects and translational aspects to produce the matrix. Unfortunately, the resulting motion matrix is hard to interpolate. More rewarding is a screw representation, but this requires specialized data structures and Chasles' theorem to compute.

The point is that these linear algebra constructions are specific for each case, and apparently tricky enough that the inventors are often remembered by name. By contrast, the geometric algebra definition of a motion operator as a ratio is easily reinvented by any application programmer.

1.2.6 CLOSED-FORM INTERPOLATION AND PERTURBATION

In many applications, one would like to apply a motion gradually or deform it continuously (for instance, to provide smooth camera motion between specified views). In geometric algebra, interpolation of motions is simple: one just applies the corresponding versor V piecemeal, in N steps of $V^{1/N}$. That N^{th} root of a motion versor V can be determined by a logarithm, in closed form, as $\exp(\log(V)/N)$. For a rotor representing a rotation at the origin, this retrieves the famous "slerp" interpolation formula of quaternions, but it extends beyond that to general Euclidean motions. Blending of motions can be done by blending their logarithms.

By contrast, it is notoriously difficult to interpolate matrices. The logarithm of a matrix can be defined but it is not elementary, and not in closed form. A straightforward way to compute it is to take the eigenvalue decomposition of the rigid body motion matrix in the homogeneous coordinate framework, and take the Nth root of the diagonal matrix. Such numerical techniques makes the matrix logarithm expensive to compute and hard to analyze.

Perturbations of motions are particularly easy to perform in geometric algebra: the small change in the versor-based motion VX/V to any element X can be simply computed as $X \times B$, the commutator product of X with the bivector logarithm of the perturbing versor. This is part of *geometric calculus*, an integrated method of taking derivatives of geometric elements relative to other geometric elements. It naturally gets into differential geometry, a natural constituent of any complete framework that deals with the geometry of physical space.

1.3 PROGRAMMING GEOMETRY

The structural possibilities of the algebra may theoretically be rich and inviting, but that does not necessarily mean that you would want to use it in practical applications. Yet we think you might.

1.3.1 YOU CAN ONLY GAIN

Geometric algebra is backwards-compatible with the methods you already use in your geometrical applications.

Using geometric algebra does not obviate any of the old techniques. Matrices, cross products, Plücker coordinates, complex numbers, and quaternions in their classical form are

all included in geometric algebra, and it is simple to revert to them. We will indicate these connections at the appropriate places in the book, and in some applications we actually revert to classical linear algebra when we find that it is more efficient or that it provides numerical tools that have not yet been developed for geometric algebra. Yet seeing all these classical techniques in the context of the full algebra enriches them, and emphasizes their specific geometric nature.

The geometric algebra framework also exposes their cross-connections and provides universal operators, which can save time and code. For example, if you need to rotate a line, and you have a quaternion, you now have a choice: you can convert the quaternion to a rotation matrix and apply that to the positional and directional aspects of the line separately (the classical method), or you view the quaternion as a rotor and apply it immediately to the line representation (the geometric algebra method).

1.3.2 SOFTWARE IMPLEMENTATION

We have made several remarks on the simpler software structure that geometric algebra enables: universal operators, therefore fewer data types, no conversions between formalisms, and consequently a simpler data flow.

Having said that, there are some genuine concerns related to the size of geometric algebra. If you use the conformal model to calculate with the Euclidean geometry of 3-D space, you use a 5-D vector space and its $2^5 = 32$ subspaces. In effect, that requires a basis of 32 elements to represent an arbitrary element. Combining two elements could mean 32×32 real multiplies per geometric product, which seems prohibitive.

This is where the actual structure of geometric algebra comes to the rescue. We will explain these issues in detail in Part III, but we can reassure you now: geometric algebra can compete with classical approaches if one uses its algebraic structure to guide the implementation.

- Elements of geometric algebra are formed as products of other elements. This implies that one cannot make an arbitrary element of the 32-dimensional framework. Objects typically have a single dimensionality (which is three for circles and lines) or a special structure (all flats contain the point at infinity ∞). This makes the structure of geometrically significant elements rather sparse. A good software implementation can use this to reduce both storage and computation.
- On the other hand, the 32 slots of the algebra are all used somehow, because they are geometrically relevant. In a classical program, you might make a circle in 3-D and would then have to think of a way to store its seven parameters in a data structure. In geometric algebra, it automatically occupies some of the $\binom{5}{3} = 10$ slots of 3-vector elements in the 5-D model. As long as you only allocate the elements you need, you are not using more space than usual; you are just using the pre-existing structure to keep track of them.

- Using linear algebra, as you operate on the composite elements, you would have had to invent and write methods (for instance, to intersect a circle and a plane). This would require special operations that you yourself would need to optimize for good performance. By contrast, in geometric algebra everything reduces to a few basic products, and their implementation can be optimized in advance. Moreover, these are so well-structured that this optimization can be automated.
- Since all is present in a single computational framework, there is no need for conversions between mathematically different elements (such as quaternions and rotation matrices). Though at the lower level such conversions may be done for reasons of efficiency, the applications programmer works within a unified system of geometrically significant elements of computation.

Using insights and techniques like this, we have implemented the conformal model and have used it in a typical ray-tracing application with a speed 25 percent slower than the optimized classical implementation (which makes it about as costly as the commonly used homogeneous coordinates and quaternion methods), and we believe that this overhead may be reduced to about 5 to 10 percent. Whether this is an acceptable price to pay for a much simpler high-level code is for you to decide.

We believe that geometric algebra will be competitive with classical methods when we also adapt algorithms to its new capabilities. For instance, to do a high-resolution rendering, you now have an alternative to using a much more dense triangulation (requiring many more computations). You could use the new and simple description of perturbations to differentially adapt rays of a coarse resolution to render an ostensibly smoother surface. Such computation-saving techniques would easily compensate for the slight loss of speed per calculation.

Such algorithms need to be developed if geometric algebra is to make it in the real world. We have written this book to raise a new generation of practitioners with sufficient fundamental, intuitive, and practical understanding of geometric algebra to help us develop these new techniques in spatial programming.

1.4 THE STRUCTURE OF THIS BOOK

We have chosen to write this book as a gradual development of the algebraic terms in tandem with geometric intuition. We describe the geometric concepts with increasing precision, and simultaneously develop the computational tools, culminating in the conformal model for Euclidean geometry. We do so in a style that is not more mathematical than we deem necessary, hopefully without sacrificing exactness of meaning. We believe this approach is more accessible than axiomatizing geometric algebra first, and then having to discover its significance afterwards.

The book consists of three parts that should be read in order (though sometimes a specialized chapter could be skipped without missing too much).

1.4.1 PART I: GEOMETRIC ALGEBRA

First, we get you accustomed to the outer product that spans subspaces (and to the desirability of the "algebraification of geometry"), then to a metric product that extends the usual dot product to these subspaces. These relatively straightforward extensions from linear algebra to a multilinear algebra (or subspace algebra) already allow you to extend linear mappings and to construct general intersection products for subspaces. Those capabilities will extend your linear algebra tool kit considerably.

Then we make the transition to true geometric algebra with the introduction of the geometric product, which incorporates all that went before and contains more beyond that. Here the disadvantage of the approach in this book is momentarily annoying, since we have to show that the new definitions of the terms from the earlier chapters are "backwards compatible." But once that has been established, we can rapidly move on from considering objects (the subspaces) to operators acting on them. We then easily absorb tools you may not have expected to encounter in real vector spaces, such as complex numbers and quaternions. Both are available in an integrated manner, real in all normal senses of the word, and geometrically easily understood.

Part I concludes with a chapter on geometric differentiation, to show that differential geometry is a natural aspect of geometric algebra (even though we will use it only incidentally in this book).

1.4.2 PART II: MODELS OF GEOMETRY

In Part II the new algebra will be used as a tool to model aspects of mostly Euclidean geometry. First, we treat directions in space, using the *vector space model*, already familiar from the visualizations used in Part I to motivate the algebra.

Next, we extend the vector embedding trick of homogeneous coordinates from practical computer science to the complete *homogeneous model* of geometric algebra, which includes Plücker coordinates and other useful methods.

Finally, in Chapter 13 we can begin to treat the *conformal model* of the motivating example in Section 1.1. The conformal model is the model that has Euclidean geometry as an intrinsic part of its structure; all Euclidean motions are represented as orthogonal transformations. We devote four chapters to its definition, constructions, operators, and its ability to describe general conformal transformations.

1.4.3 PART III: IMPLEMENTATION OF GEOMETRIC ALGEBRA

To use geometric algebra, you will need an implementation. Some are available, or you may decide to write your own. Naïve implementations run slow, because of the size of the algebra (32-D for the basis of the conformal model of a 3-D Euclidean space).

In the third part of this book, we give a computer scientist's view of the algebraic structure and describe aspects that are relevant to any efficient implementation, using its multiplicative and sparse nature. We end with a simple ray tracer to enable comparison of computational speeds of the various methods in a computer graphics application.

1.5 THE STRUCTURE OF THE CHAPTERS

Each regular chapter consists of an explanation of the structure of its subject. We explain this by developing the algebra with the geometry, and provide examples and illustrations. Most of the figures in this book have been rendered using our own software package, `GAViewer`. This package and the code for the figures are available on our web site,

http://www.geometricalgebra.net.

We recommend that you download the software, install it, and follow the instructions to upload the figures. You can then interact with them. At the very least you should be able to use your mouse for 3-D rotation and translation to get the proper spatial impression of the scene. But most figures also allow you to interactively modify the independent elements of the scene and study how the dependent elements then change.

You can even study the way we have constructed them[1] and change them on a command line; though if you plan to do that we suggest that you first complete the `GAViewer` tutorial on the web site. This will also allow you to type in formulas of numerical examples.

If you really plan to use geometric algebra for programming, we recommend doing the drills and programming exercises with each chapter. The programming exercises use a special library, the *GA sandbox*, also available from the web site. It provides the basic data structures and embedding of the products so that you can program directly in geometric algebra. This should most closely correspond to how you are likely to use it as an extension of your present programming environment.

We also provide structural exercises that help you think about the coherence of the geometric algebra subject in each chapter and ask you to extend some of the material or provide simple proofs. For answers to these exercises, consult the web site.

Historically, geometric algebra has many precursors, and we will naturally touch upon those as we develop our concepts and terms. We do not meticulously attribute all results and thoughts, but occasionally provide a bit of historic perspective and contrast with traditional approaches. At the end of each chapter, we will give some recommended literature for further study.

---

1 But be warned that some illustrative figures of the simpler models may use elements of the conformal model in their code, since that is the most natural language to specify general elements of Euclidean geometry.