GABLE: A Matlab Tutorial for Geometric Algebra

Leo Dorst, Stephen Mann, and Tim Bouma

December 3, 2002

Abstract

In this tutorial we give an introduction to geometric algebra, using our Matlab package GABLE (Geometric AlgeBra Learning Environment). In the geometric algebra for 3-dimensional Euclidean space, we graphically demonstrate the ideas of the geometric product, the outer product, and the inner product, and the geometric operators that may be formed from them. We give several demonstrations of computations you can do using the geometric algebra, including projection and rejection, orthogonalization, interpolation of rotations, and intersection of linear offset spaces such as lines and planes. We emphasize the importance of blades as representations of subspaces, and the use of `meet` and `join` to manipulate them. We end with Euclidean geometry of 2-dimensional space as represented in the 3-dimensional homogeneous model.

Contents

1 Introduction

This is an introduction to *geometric algebra*, which is a structural way to think about and calculate with geometry. It differs in its nature from linear algebra, constructive Euclidean geometry, differential geometry, vector calculus and other ways you may have learned before; yet it encompasses them all in a natural manner, including some extra things like complex numbers and quaternions. To help understand and visualize the geometry, we have implemented GABLE (Geometric AlgeBra Learning Environment) in Matlab, which we use in this tutorial to illustrate our examples.

We believe geometric algebra is going to be useful to all of us applying geometry in our problems in robotics, vision, computer graphics, etcetera. This tutorial is meant to be an easily accessible introduction that gives you an overview of the subject, in a way that helps you assess its power, and helps you decide whether to study it seriously.

There are several reasons why geometric algebra is so convenient to work with in geometry, and they all involve the capability to talk constructively about important geometrical concepts (we name them below), which are all embedded as elementary objects in the algebra. Having those available will change your thinking in a strangely powerful way, and geometric algebra then provides the computational tools to implement that new thinking. Obviously, you can't change your thinking overnight, and so we will demonstrate some of it in the tutorial to give you a flavor.

Here are some teasers to get you interested:

- *subspaces and dependence* (Section 2.2.6)
  Subspaces become elementary objects; $\mathbf{a} \wedge \mathbf{b}$, for instance, is an object that represents the plane spanned by vectors $\mathbf{a}$ and $\mathbf{b}$, and $\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}$ the volume spanned by vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$. Linear dependence is then easily expressible: $\mathbf{a} \wedge \mathbf{b} = 0$ implies that $\mathbf{a}$ and $\mathbf{b}$ are dependent since they do not span a plane.
- *division by subspaces* (Section 2.4.2)
  In geometric algebra, you can divide by vectors, planes, etcetera. This makes solving equations between geometric objects easier; and it allows interesting coordinate-free construction of geometric relationships. For instance, the component of a vector $\mathbf{x}$ perpendicular to a plane $\mathbf{a} \wedge \mathbf{b}$ is the volume spanned by $\mathbf{x}$, $\mathbf{a}$ and $\mathbf{b}$, divided by the plane. In formula: $(\mathbf{x} \wedge \mathbf{a} \wedge \mathbf{b})(\mathbf{a} \wedge \mathbf{b})^{-1}$.
- *parameterization and duality* (Section 2.4.3)
  It is often convenient to represent objects dually: planes by normal vectors, lines by their slope and intercept, etcetera. Mathematicians have told us that these dual objects live in dual spaces (the dual representation of a plane is *not* a vector but a 1-form, and transforms as such), and this makes their representation a mapping between spaces. In geometric algebra, objects and their duals live in the same algebra, and are algebraically related: 'dualization' of an object is simply 'dividing by the volume element' of the space it lives in. This has enormous advantages, since this transition to a dual description does not involve a change of space and data structures.
- *operations are products of vectors* (Section 3.4)
  In geometric algebra, the ratio $\mathbf{b}/\mathbf{a}$ of two vectors defines the rotation/scaling between them, in all its aspects: both the plane it happens in, and the angle between them, and the dilation (scale) factor. Such characterizations of operations are easy to compose, and can be applied not only to rotate vectors, but (using the same formula) also planes, volumes, etcetera, in n-dimensional space.
- *complex numbers and quaternions* (Section 3.6)
  Have you ever wondered why quaternions work and what they are? In geometric algebra, we will derive them naturally, and they will not be anything worth a special term. And it will be clear how they generalize to describe rotations in n dimensions. They are just an example of the many efficient structures present in geometric algebra that pop up as you use them. Complex numbers, which describe rotations in a plane, are another example. We will find that every plane $\mathbf{a} \wedge \mathbf{b}$ in Euclidean space has a 'complex number system' associated with it, and that this is basically the $\mathbf{b}/\mathbf{a}$ we mentioned above as the rotation operator for that plane. All these things are connected in a highly satisfying manner (as we hope you will agree when you're done).
- *Euclidean geometry* (Section 5.2)
  Euclidean geometry, with its offset lines and triangles, their lengths, intersections, perpen-

diculars etc., is naturally treated in geometric algebra; the trick is to find an embedding of this geometry into the geometric algebra of a nice space. Geometric algebra can do this by generalizing 'homogeneous coordinates'. Vectors represent offset points, planes represent offset lines etcetera; and in geometric algebra these are all elementary objects of computation. (In chapter 5, we will illustrate this for 2-dimensional space.)

- `meet` (Section 5.4)
  There is a powerful operation called the *meet*, which is the general incidence relationship between geometric entities. The `meet` of two lines in 3D, for instance, will return the *intersection point* and intersection strength (sine of angle between the lines) if the lines intersect; but it will return a *line* if they coincide, and it will return *the Euclidean distance* between them if they do not have a point in common. Using geometric algebra, we are capable of defining such operators without forcing the user to split them into cases.
- *geometric differentiation*
  Something we will not be able to cover in this tutorial, but which is important to applications in continuous geometry, is the capability to differentiate and integrate with respect to geometric objects. It becomes possible to find the optimal orientation explaining a set of measurement data by a standard optimization procedure: define the criterion that you want to optimize, *differentiate with respect to rotation* and set this equal to zero to find the extremum. Many techniques now become transportable from ordinary optimization theory of functions to the optimization of geometrical objects.

You may find in this tutorial lots of things that are familiar, because a lot of this work has been invented before in other contexts than geometric algebra. It is only recently that we understand how it all fits together into one framework, and how important that is for the computer sciences. It now becomes possible to write one book, and one computer program, which contains all the geometry we might ever need. Gone would be the transitions between parts of the real-world problem that are solved by linear algebra, vector calculus, or differential geometry, with the accompanying inefficiency and sensitivity to bugs. All would just be done within the framework of geometric algebra.

In this tutorial, we introduce terms gradually, and give you a geometric intuition of their meaning as we go along. We limit almost all of our explanations to the geometric algebra of Euclidean 3-dimensional space, which is denoted $\mathcal{C}\ell_{3,0}$. Other geometric algebras are important to computer science as well, but in them intuition is somewhat harder to obtain, and that is the reason we decided not to use them in this introductory tutorial.

1.1 Getting started

To get the most out of reading this tutorial, you should read it while running Matlab on a color display, and try the sample code and exercises. The tutorial was developed on Matlab 5.3; it also runs on Matlab 5.2, but it will not run on Matlab 5.1. It has been tested on both Sun workstations and on IBM PCs.

This tutorial is not a tutorial on Matlab, and to work more easily with it you should probably read some introduction into Matlab before using our GABLE package. Yet you should be able to run most of the demos described in this tutorial with no supporting documentation; on the other hand, you will need to know more about Matlab to perform the exercises.

You can download the software from one of the two following web pages:

```
http://www.wins.uva.nl/~leo/GABLE/
http://www.cgl.uwaterloo.ca/~smann/GABLE/
```

The instructions there will tell you how to set up the software. You will have to install Matlab separately.

Assuming you install GABLE in a directory called `gable`, then to use it, add to your Matlab path with the `addpath('.../gable')` command, where `...` is the directory path to the `gable` directory. You may also wish to put this command in your initialization file. See the Matlab documentation for details on both subjects.

You can get a quick introduction to the basics of geometric algebra by running the Matlab/GABLE command `GAdemo`. This demonstration routine will show you vectors, bivectors, and trivectors, as well as introduce the three products of geometric algebra. However, `GAdemo` is not a substitute for reading this tutorial, as the interpretations and description of how to use the geometric algebra is too involved for the demo script. Thus, after running `GAdemo` you will need to read the remainder of this tutorial.

1.2 Notation

In this tutorial, we will use standard, italic math fonts for our equations. When giving Matlab/GABLE code, or specifying Matlab variables in our text, we will use typewriter font. We will elaborate on some further parts of our notation as we introduce it. Further, in our Matlab/GABLE code samples, unless otherwise specified, we will assume that you clear the graphics window (using `clf`) before running each code fragment. If we have a running example (i.e., where the sample code is separated with explanatory text), the later code fragments will begin with

```
>> %...
```

to indicate that this is a continuation of the previous code fragment (you should not type the '`...`' in Matlab!). We may denote the variables you need to continue from the previous segment. E.g.,

```
>> %... needs X
```

means that the example is continued from the previous fragment, but that you only need the variable `X` from that fragment. Occasionally, we will put comments in our code fragments to indicate what the code is doing; such comments look like

```
>>    % === words
```

You do not need to type such comments into Matlab, and in general you do not need to type anything following a percent sign on a line.

Sometimes an illustration involves a lot of typing. To save you typing, we have put this code in a routine called `GAblock`. Any code sequence that appears in a `GAblock` will be prefaced by

```
>> %GAblock(N)
```

where `N` denotes the appropriate section within `GAblock`. To run this sequence, type everything after the '`%`' sign on that line. The running of such a code sequence will stop on any line with a '`%%`'. At such times, we want you to see something, and give you a special prompt:

```
GAblock >>
```

At this prompt, you may type Matlab/GABLE commands; when done, just press return and the block of code will resume running. For a continued code fragment (i.e., one that starts with `%...`'), you will be prompted to read the tutorial before continuing. To quit a `GAblock` sequence early, use `GAend`.

Note that we insert the `GAblock` prompts for a reason: either you should be understanding something in the picture, or you need to understand a result on the screen. At these prompts, you can and should type Matlab/GABLE code to test things, to rotate the view on the screen, etc., until you understand what is being illustrated.

By the way, another way to save typing in some of the more repetitive exercises is the standard Matlab feature to use the up-arrow to step through your command history, permitting you to change earlier commands by 'inline editing': overtyping, deletion and insertion.

2 The products of geometric algebra

This section introduces the basics of the geometric algebra, and gives the GABLE commands for performing the operations. Many of the objects have a graphical representation, and can be displayed on the screen using the `draw` command. Table 1 summarizes the GABLE commands described in this section and in the rest of this tutorial. We will introduce these routines gradually, as we need them; the table is here just for reference. Some additional commands and details on some of the commands in this table can be found in the appendices. You can also get a summary of commands using the Matlab '`help gable`' command (assuming you have set up the Matlab path correctly).

2.1 Scalar product

The defining elements of geometric algebra are the vectors of *a linear space of vectors over scalars.* In our package, we use an orthonormal basis to represent this linear space, with vectors `e1`, `e2`, and `e3`, and we will always use integer or real numbers as our scalars.[1] You can scale

[1]Frames are a necessary crutch for input and output; most of our computations will be independent of the frame of representation, in the sense that our equations can be expressed in a coordinate-free manner. For a further discussion of frames, see Appendix B.

Command	**Arguments**	**Result**
`e1,e2,e3`		Basis vectors for generating the geometric algebra
`I3`		The unit pseudoscalar
`+,--,*,/,^`	multivectors	The operations of our geometric algebra
`dual`	(multivector)	Compute the dual of a multivector
`gexp`	(multivector)	The geometric product exponential, e^{mv}
`grade`	(blade)	Return the grade of a blade (-1 if a multivector)
	(multivector,n)	Return the portion of the multivector that is of grade n
`inner`	(mv,mv)	Take the inner product of two multivectors
`inverse`	(multivector)	Compute the inverse (if it exists) of the multivector
`isGrade`	(multivector,g)	Test if multivector is blade of grade g
`join`	(multivector)	Join two blades
`meet`	(multivector)	Meet two blades
`norm`	(multivector)	Returns the norm of a multivector.
`sLog`	(spinor)	The geometric logarithm of a spinor
`unit`	(blade)	Returns the parallel blade of unit length and same sign.
*`draw`	(A)	draw object A
`clf`		clears the screen
`GAview`	([az el])	Set the view; similar to Matlab 'view'
`GAorbiter`		Rotates the view; optional arguments give angle and time
`GAbvShape`	(TYPE)	With no arguments, return the shape of bivector
		With one argument (TYPE='default', 'Dutch', 'Canadian', or 'American') set the bivector shape to this type
*`DrawBivector`	(v1,v2)	Draw bivector spanned by v1,v2 as parallelogram
`DrawTrivector`	(v1,v2,v3)	Draw trivector spanned by v1,v2,v3 as parallelepiped
`DrawOuter`	(A,B)	Draw the outer product between A and B
`DrawInner`	(A,B)	Draw the inner product between A and B
`DrawGP`	(A,B)	Draw the geometric product between A and B
*`DrawPoint`	(P)	Draw a point on the tips of a cell array of GA vectors
`DrawPolyline`	(P)	Draw a polyline on the tips of a cell array of GA vectors
*`DrawPolygon`	(P)	Draw a polygon on the tips of a cell array of GA vectors
*`DrawSimplex`	(S)	Draw a simplex given as a cell array of GA vectors
*`DrawHomogeneous`	(S)	Draw a simplex given in homogeneous representation

Table 1: Table summarizing geometric algebra Matlab commands of GABLE. '`*`'-commands take optional color arguments.

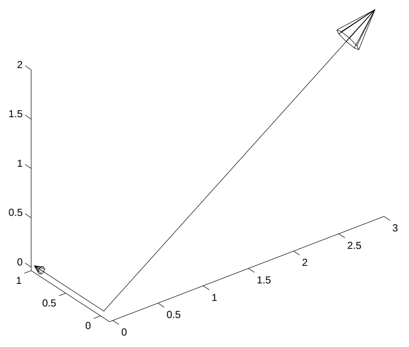

Figure 1: `draw(3*e1+2*e3); draw(e2)`

and add these vectors in the usual manner. For example, to create the vector $\mathbf{a} = 3\mathbf{e}_1 + 2\mathbf{e}_2$, you would type

```
>> a = 3*e1+2*e3
ans =
      3*e1 + 2*e3
```

Do *not* use spaces in the definition; the particular way we have overloaded the Matlab products may produce the rather cryptic error "`Too many input arguments`". Also, do not forget the '`*`' for the product with a scalar, or Matlab will interpret '`3e1`' as $3 \times 10^1 = 30$.

The `norm` of a vector is its length (in the standard metric in Euclidean space):

```
>> ... needs a
>> norm(a)
ans =
    3.6056
```

You can draw a vector (and many of the geometric objects) using the command `draw`. To draw the above vector, type

```
>> draw(a)
```

In the graphics window, you will see a line with an arrow head. If we draw a second vector,

```
>> draw(e2)
```

we see the space get rescaled so that both vectors appear in the same plot. Your screen should now look something like Figure 1. Note that both vectors start at the origin and have their arrow head at the end of the line segment away from the origin. If you would like to understand the spatial relationship better, type:

```
>> GAorbiter
```

and the plot will turn 360 degrees over 10 seconds. You can get smaller rotations by giving it an argument; try `GAorbiter(180)`. If you give a second argument, you can change the time over which it rotates; try `GAorbiter(180,5)`. In later versions of Matlab (beyond 5.3) the figure window contains a button to help you rotate a figure interactively.

You can also draw a scalar, though that is not very exciting:

```
>> draw(2)
```

This prints the scalar above any plot you might have made.

2.2 The outer product

2.2.1 Definition

Geometric algebra has an *outer product*, often called a *wedge product*. The outer product has the properties of *anti-symmetry*, *linearity* and *associativity*. For vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ we have:

- $\mathbf{v} \wedge \mathbf{w} = -\mathbf{w} \wedge \mathbf{v}$, so that $\mathbf{v} \wedge \mathbf{v} = 0$
- $\mathbf{u} \wedge (\mathbf{v} + \mathbf{w}) = \mathbf{u} \wedge \mathbf{v} + \mathbf{u} \wedge \mathbf{w}$

- $\mathbf{u} \wedge (\mathbf{v} \wedge \mathbf{w}) = (\mathbf{u} \wedge \mathbf{v}) \wedge \mathbf{w}$

The outer product of a vector $\mathbf{v}$ with a scalar α, or of two scalars α and β, we define to be equal to the scalar product

$$\alpha \wedge \beta = \alpha\beta \quad \text{and} \quad \alpha \wedge \mathbf{v} = \alpha\mathbf{v},$$

and then use associativity to extend that to the evaluation of more involved terms.

(The properties of the outer product of two vectors are similar to the properties of the *cross product* for two vectors in 3D. Yet it results in a different geometrical object, as we will soon see. We will discuss the correspondence between the two in Section 2.4.3).

2.2.2 Bivectors

In GABLE, the circumflex symbol (ˆ) is used to take the outer product of objects. E.g., the outer product of `e1` and `e2` is formed by `e1^e2`. Here are some to try, and you may want to verify the results on paper using the definition:

```
>> 2^e2
>> e1^e2
>> e2^e1
>> e1^(e1+e2)
>> (e1+e2)^(e1-e2)
```

The outcome of the wedge product of two vectors thus contains terms like $\alpha(\mathbf{e}_1 \wedge \mathbf{e}_2)$, etc., which can not be simplified further to vectors or scalars. This outcome is therefore a *new* kind of object in our geometric algebra, called a *bivector.*

As you try more combinations, you find that *any* bivector can be expressed as a scalar-weighted linear combination of the standard bivectors $\mathbf{e}_1 \wedge \mathbf{e}_2$, $\mathbf{e}_2 \wedge \mathbf{e}_3$, $\mathbf{e}_3 \wedge \mathbf{e}_1$, formed by outer product between the vectors in the vector basis. This follows easily from the linearity and anti-symmetry properties in the definition of the outer product. These bivectors thus form a *bivector basis* (but as in the case of a basis for vectors, other bases may serve just as well). Algebraically, the set of bivectors in 3-dimensional space is therefore in itself a 3-dimensional linear space.

You can view a bivector as a *directed area element*, directed both in the sense of specifying a plane and an orientation in that plane. In general, with ϕ denoting the angle from $\mathbf{u}$ to $\mathbf{v}$ and with $\mathbf{i}$ the unit directed area of the $(\mathbf{u}, \mathbf{v})$-plane, we can write:

$$\mathbf{u} \wedge \mathbf{v} = |\mathbf{u}|\,|\mathbf{v}|\,\sin(\phi)\mathbf{i}.$$

You recognize that $|\mathbf{u}|\,|\mathbf{v}|\,\sin(\phi)$ is the directed area spanned by $\mathbf{u}$ and $\mathbf{v}$; and as $\mathbf{u}$ and $\mathbf{v}$ get more parallel, this quantity becomes smaller. As you make the angle negative, the bivector becomes negative (in agreement with the anti-symmetry of the outer product since now $\mathbf{u}$ and $\mathbf{v}$ have switched roles); this is what we mean by a *directed* area element. The $\mathbf{i}$ indicates the plane in which this takes place; it is therefore a geometric 'unit direction of dimension 2' of what is measured.

Graphically we represent the bivector in our package as a directed circle in the bivector plane, with arrows along its border to indicate the orientation. The area of the circle is the magnitude of the bivector. For example, let us draw some vectors, and then draw a bivector:

```
>> clf; draw(2*e1+e3); draw(e2);
>> draw(e1^e2)
```

You should see in the graphics window something like Figure 2. Perform `GAorbiter` to appreciate the spatial relationships better: note that the circle lies in the plane containing both `e1` and `e2`.

Another way of appreciating what goes on is to draw the inputs and output of the outer product separately, which can be done by the command `DrawOuter()`:

```
>> clf;
>> B = DrawOuter(e1,e2)
ans =
     e1^e2
>> GAorbiter
```

This gives two spaces, with in blue and green the two arguments of the product (in that order), and in red the result. You can place the result in the first space for comparison, by:

```
>> %...
>> subplot(1,2,1);
>> draw(B,'r');
```

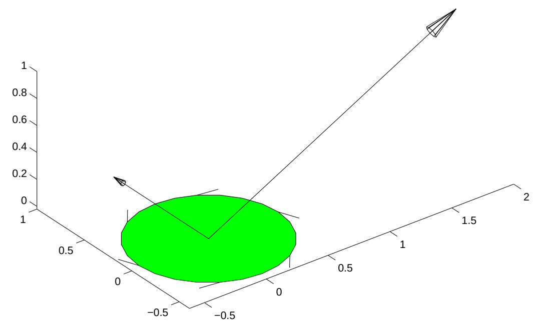

Figure 2: `draw(2*e1+e3); draw(e2); draw(e1^e2)`

Try the outer product on `e1^e1`, `e1^e2`, `e2^e1`, and `e1^(e1+e2+e3)`, using `draw()` (use the optional color argument to tell the results apart) or `DrawOuter()` to draw any non-scalar results. You may want to use `clf` to clear the screen between evaluations.

The `norm` of a bivector is the absolute value of the area it represents:

```
>> norm((e1+e2)^e3)
ans =
    1.4142
```

Warning: In one case, our implementation of the outer product gives the wrong answer, namely if you perform the outer product of two scalars: `2^3` gives `8` (wrong!) rather than `6` (correct!), since we were not able to overload the exponentiation operator in Matlab for scalars. So when you want to multiply scalars, you will have to use `*`, as in `2*3`. See Appendix A.3 for a further discussion of this problem.

2.2.3 Trivectors

Taking the outer product of three vectors yields yet another object, which is naturally called a *trivector*. It is a directed volume element. In 3-dimensional space, all such elements must be multiples of the unit directed volume element, which we denote by $\mathbf{I}_3$. (In other words, algebraically the trivectors of a 3-dimensional vector space form a 1-dimensional linear space with basis $\mathbf{I}_3$.) In an orthonormal basis $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ for our Euclidean 3-space, we equate it with the volume spanned by the 'right-handed' frame: $\mathbf{I}_3 \equiv \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$. The unit directed volume $\mathbf{I}_3$ is often called the *(unit) pseudoscalar* of 3-dimensional Euclidean space.

We have implemented $\mathbf{I}_3$ as `I3`. Verify the outcome of the following expressions by hand, to get some dexterity in manipulations with the wedge product on the basis of its definition:

```
>> e1^(e2^e3)
>> (e1^e2)^e3
>> e1^e2^e3
>> e3^e2^e1
>> e1^(e1+2*e2)^e3
>> e1^(e1+2*e2)^e1
>> norm(e1^e2^e3)
```

Notice that the trivector $\mathbf{e}_3 \wedge \mathbf{e}_2 \wedge \mathbf{e}_1$ equals $-\mathbf{I}_3$: these vectors in this order form a 'left-handed' frame. The terms denoting 'handedness' are therefore not explicit conventions anymore, they have become part of the computational framework of geometric algebra as the signs of trivectors. The `norm` of a trivector is absolute value of the volume; if you need a signed scalar denoting the volume of a trivector $\mathbf{T}$, use $\mathbf{T}/\mathbf{I}_3$.

Conceptually, a trivector represents an oriented volume. Graphically, we represent it by a *sphere*, which we render as a line drawing. The magnitude of the trivector is represented by the volume of the sphere. To indicate the orientation, we draw line segments emanating from the points on the sphere; the orientation is indicated by whether these line segments go into or out of the sphere. Try

```
>> draw(I3,'g')
>> draw(-0.5*I3,'r')
```

Note that it is unfortunately difficult to visualize more than one pseudoscalar using this representation as it makes the drawing too cluttered. Also note that although we represent pseudoscalars with a line drawing of the surface of the sphere, the pseudoscalar is actually a *solid* volume element.

2.2.4 Quadvectors?

If you try taking some outer products of four or more vectors, you will find that these are all zero. You may understand why this should be: since only three vectors in 3-space can be independent, any fourth must be writable as a weighted sum of the other three; and then the anti-symmetry of the outer product kills any term in the expansion. For instance:

$$\begin{aligned}
&\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge (\mathbf{e}_1 + \mathbf{e}_2) \\
&= \quad \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_1 + \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_2 \\
&= \quad (\mathbf{e}_1 \wedge \mathbf{e}_1) \wedge \mathbf{e}_2 \wedge \mathbf{e}_2 - \mathbf{e}_1 \wedge (\mathbf{e}_2 \wedge \mathbf{e}_2) \wedge \mathbf{e}_3 = 0.
\end{aligned}$$

So the highest order object that can exist in a 3-dimensional space is a trivector. But you can also see that this is not a limitation of geometric algebra in general: if the space had more dimensions, the outer product would create the appropriate hyper-volumes.

If all we are interested in is planar geometry, then all vectors can be written as the linear combination of two basis vectors, such as $\mathbf{e}_1$ and $\mathbf{e}_2$; in that case, the highest order object would be a bivector. We would then call $\mathbf{I}_2 \equiv \mathbf{e}_1 \wedge \mathbf{e}_2$ a pseudoscalar of that 2-dimensional space. In n-dimensional space, *the pseudoscalar is the highest dimensional object in the space.* It received this rather strange name because it is 'dual' to a scalar, as we will see in Section 2.4.3.

2.2.5 0-vectors

In the same vein of the interpretation of k-vectors as geometrical k-dimensional subspaces based in the origin, we can reinterpret the scalars geometrically. Since these are 0-vectors, they should represent a 0-dimensional subspace at the origin, i.e. *geometrically, a scalar is a weighted point at the origin.* This is a fully admissible geometric object, and therefore it should not be surprising that it is a member of the basis $\{1, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_2 \wedge \mathbf{e}_3, \mathbf{e}_3 \wedge \mathbf{e}_1, \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3\}$ of the geometric algebra of 3-dimensional space.

2.2.6 Use: parallelness and spanning subspaces

The outer product of two vectors $\mathbf{u}$ and $\mathbf{v}$ forms a bivector. When you keep $\mathbf{u}$ constant but make $\mathbf{v}$ increasingly more parallel to it (by turning it in the $(\mathbf{u}, \mathbf{v})$-plane), you find that the bivector remains in the same plane, but becomes smaller, for the area spanned by the vectors decreases. When the vectors are parallel, the bivector is zero; when they move beyond parallel ($\mathbf{v}$ turning to the 'other side' of $\mathbf{u}$) the bivector reappears with opposite magnitude. Try this:

```
>> e1^e2
>> e1^unit(e1+e2)
>> e1^unit(10*e1+e2)
>> e1^unit(1000*e1+e2)
>> e1^e1
>> e1^unit(1000*e1-e2)
>> e1^unit(10*e1-e2)
```

(`unit` returns a vector parallel and of the same orientation as its argument, but of unit length.)

A bivector may therefore be used as a measure of parallelness: in Euclidean space $\mathbf{u} \wedge \mathbf{v}$ equals zero if and only if $\mathbf{u}$ and $\mathbf{v}$ are parallel, i.e., lie on the same 1-dimensional subspace. Note that this even holds for 1-dimensional space.

Similarly, a trivector is zero if and only if the three vectors that compose it lie in the same plane (2-dimensional subspace). We then do not call them 'parallel'; the customary expression is 'linearly dependent', but the geometric intuition is the same. If the vectors are 'almost' in the same plane, they span a 'small' trivector (relative to their norms times the unit pseudoscalar).

In fact, we can use a bivector $\mathbf{B}$ to represent a plane through the origin:

$$\text{vector } \mathbf{x} \text{ in plane of } \mathbf{B} \iff \mathbf{x} \wedge \mathbf{B} = 0.$$

And this $\mathbf{B}$ even represents a *directed plane*, for we can say that a point $\mathbf{y}$ is at the 'positive side' of the plane if $\mathbf{y} \wedge \mathbf{B}$ is a positive volume, i.e., a positive multiple of the unit pseudoscalar $\mathbf{I}_3$. We will come back to this powerful way of representing planes later (Section 3.7); but for now you understand why we like to think of a bivector as a directed plane element.

2.2.7 Blades and grades

We now have all the basic elements for our geometric algebra of 3-space: scalars, vectors, bivectors, and trivectors (pseudoscalars). We have constructed each of these from vectors and scalars using the outer product. There are some useful terms to describe this construction.

A *blade* is an object that can be written as the outer product of vectors. For instance, $\mathbf{e}_1$, or $\mathbf{e}_1 \wedge (\mathbf{e}_1 + 2\mathbf{e}_2)$, or $\mathbf{I}_3 \equiv \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$.

The *grade* of a blade is the dimension of the subspace that the blade represents. So for instance $\text{grade}(\mathbf{e}_1 \wedge (\mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3)) = 2$, and $\text{grade}(\mathbf{I}_3) = 3$. As you see, the outer product of a object with a vector raises the grade of the object by one (or gives 0).

We can make general objects in our algebra by taking *scalar-weighted sums of blades* such as $1 + \mathbf{e}_1 + \mathbf{e}_2 \wedge \mathbf{e}_3$. Such objects are called *multi-vectors*. In this construction, a blade is called an *m-vector*, with m the grade of the blade. In that sense, a scalar is a 0-vector. Often such a multivector is of mixed grade (do not worry about its geometrical interpretation yet).

In GABLE, we have a routine `grade` that when given an object, returns the grade of that object. If the object is of mixed grade, `grade` returns `-1`. When invoked with a geometric object and an integer, `grade` returns the portion of the geometric object of that grade. For example,

```
>> grade(e1+I3,1)
ans =
      e1
>> grade(e1+I3,3)
ans =
      I3
```

To test if an object is of a particular grade, use the `isGrade` command:

```
>> isGrade(e1^e2,1)
ans =
      0
>> isGrade(e1^e2,2)
ans =
      1
```

In this context of blades and grades, there is a peculiarity of 3-dimensional space that does not carry over to higher dimensions: in 3-space (or 2-space, or 1-space), any multivector that is not of mixed grade can be factored into a blade. For example, we can rewrite $\mathbf{e}_1 \wedge \mathbf{e}_2 + \mathbf{e}_2 \wedge \mathbf{e}_3$ as $(\mathbf{e}_1 - \mathbf{e}_3) \wedge \mathbf{e}_2$. The former is the sum of two bivectors (and thus not in blade form), while the latter is the outer product of two vectors (and is therefore obviously a blade). In 4-dimensional space, this fails: $\mathbf{e}_1 \wedge \mathbf{e}_2 + \mathbf{e}_3 \wedge \mathbf{e}_4$ cannot be rewritten as the outer product of two vectors.

2.2.8 Other ways of visualizing the outer product

The interpretation of the bivector as a directed circle, which we have used so far, is not what everyone uses to visualize them. The standard interpretation works directly with the outer product. If you have $\mathbf{e}_1 \wedge (\mathbf{e}_1 + \mathbf{e}_2)$, then for the standard interpretation, we construct the parallelogram having $\mathbf{e}_1$ and $\mathbf{e}_1 + \mathbf{e}_2$ as two of its sides. Graphically, we would draw the vector $(\mathbf{e}_1 + \mathbf{e}_2)$ starting from the head of $\mathbf{e}_1$. The area of this parallelogram is the area of the bivector, and the two vectors give the orientation.

You can see this interpretation of the bivector in GABLE using by typing

```
>> DrawBivector(e1,e1+e2);
```

The first vector is drawn in blue, the second in green, and the area is shaded in yellow. Note, however, that the particular vectors used to construct the bivector are not unique. Any two vectors in this plane that form a parallelogram of the same directed area give the same bivector. For example,

```
>> %...
>> DrawBivector(e1,e2);
```

will draw a second parallelogram (a square) that overlaps the first parallelogram. Although these are two different parallelograms, they are coplanar and have the same directed area, therefore *they represent the same bivector.* (If you are having trouble seeing `e1^e2`, you may wish to run `clf` and rerun the second `DrawBivector` command.)

Using GABLE, you can test that these represent the same bivector by using the equality operator:

```
>> e1^(e1+e2) == e1^e2
ans =
     1
```

In Matlab, a result of '1' for a Boolean test means "true" and a result of '0' means "false." In our example, this means the geometric objects are the same, which we can also prove algebraically:

$$\mathbf{e}_1 \wedge (\mathbf{e}_1 + \mathbf{e}_2) = \mathbf{e}_1 \wedge \mathbf{e}_1 + \mathbf{e}_1 \wedge \mathbf{e}_2 = \mathbf{e}_1 \wedge \mathbf{e}_2$$

Note that the area is oriented. In particular, if we reverse the order of the vectors in the outer product, we get a different result:

```
>> e1^e2 == e2^e1
ans =
     0
```

However, they only differ by a sign:

```
>> e1^e2 == -1*e2^e1
ans =
     1
```

This is a result of the bivector representing a directed area: if we reverse the order of the arguments, then we get the opposite direction.

More generally, we can think of the bivector as representing any directed area within some simple, closed, directed curve. (To prove this we would need to develop a calculus; we will not do so in this introduction, so please just accept this statement.) We used a circle for our representation since we often will not have the creating vectors for a bivector. Indeed, depending on how we constructed the bivector such vectors may not exist, for example when we take the dual (Section 2.4.3) of a vector. While we may use any closed curve of the appropriate area, the circle is the closed curve with perfect symmetry. In our rendering of the bivector, the area of the circle indicates the area of the bivector; to indicate the orientation of the area, we draw arrows along its rim.

In a similar manner, we can view a trivector as a parallelepiped. Clear the screen, then run

```
>> DrawTrivector(e1,e1+e2,e3);
>> GAorbiter
```

to see the parallelepiped constructed for a trivector. You may find it easier to visualize if you also run `axis off` before the `GAorbiter` command.

2.2.9 Summary

In this section we have seen the outer product, which combines elements of geometric algebra to form higher dimensional elements. In particular, the outer product of two vectors is a directed area element that spans the space containing those two vectors. When applied to other blades of our space, we get higher dimensional directed volume elements.

If the vectors we combine with the outer product are linearly dependent, then the result of the outer product will be zero; if they are 'almost linearly dependent' in the sense of almost aligned, the outer product will be small. Therefore the outer product provides a quantitative and computational way to treat linear dependence.

Exercises

1. Draw (using `draw`) in different colors the bivectors $\mathbf{e}_1 \wedge \mathbf{e}_2$, $\mathbf{e}_2 \wedge \mathbf{e}_3$, and $\mathbf{e}_3 \wedge \mathbf{e}_1$.
2. Redraw the bivectors of the previous exercise using `DrawBivector`. Based on this and the previous exercise, do you have a feeling that these bivectors are orthogonal?
3. Draw using `DrawBivector` the bivectors $\mathbf{e}_1 \wedge \mathbf{e}_2$ and $\mathbf{e}_1 \wedge (\mathbf{e}_2 + 3\mathbf{e}_1)$. From this picture, do you get a feeling whether or not these two bivectors are equal? Test their equality by comparing them with the `==` operator.

4. Draw (using `draw`) the bivectors $\mathbf{e}_1 \wedge \mathbf{e}_2$ and $\mathbf{e}_2 \wedge \mathbf{e}_1$. Are these two bivectors the same? Hint: Notice how the arrows point in opposite directions. Repeat this exercise using `DrawBivector` and comment on the results.
5. Draw using both `draw` and `DrawTrivector` the trivector $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$. Now clear the screen and draw the trivector $\mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_2$ with both drawing routines. Which graphical representation gives you a better feel for orientation?
6. Draw the vectors $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$. Next draw (using `draw`) the bivector $\mathbf{e}_1\wedge\mathbf{e}_2$ and the trivector $\mathbf{e}_1\wedge\mathbf{e}_2\wedge\mathbf{e}_3$. From this picture, is it easy to tell that the drawn vectors were used to create the bivector and trivector? Redo this exercise using `DrawBivector` and `DrawTrivector`. Which graphical representation gives you a better feel for the bivector/trivector that gets created from particular vectors?
7. As we mentioned, a bivector does not really have a fixed shape. To drive this home, we have provided three alternative shapes: American, Canadian, and Dutch. To use one of these shapes, type `GAbvShape('Dutch')` for example. Then any future drawings of bivectors (using the `draw` command) will appear with the Dutch shape. You may also define your own shape using the command `GAbvShape('UserDefined',X,Y)`, where `X` and `Y` are the coordinates of the vertices of the polygonal shape you wish to use. (For these shapes, it was hard to denote the direction of the bivector by arrows along the border, so we have used a perpendicular vector related to the direction by the right-hand rule.) Play around with them to wean yourself from the habit of mentally associating a bivector with a fixed shape.

2.3 The inner product

2.3.1 Definition

In a Euclidean vector space, you may have used an *inner product*, often called the *dot product*, to quantify lengths and angles. Geometric algebra also has an inner product, and in our specific geometric algebra, $\mathcal{C}\ell_{3,0}$, the inner product on two vectors is the same as the Euclidean inner product of two vectors.

On vectors, the inner product of our geometric algebra has the standard properties of symmetry and linearity:

- $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$
- $(\alpha\mathbf{u} + \beta\mathbf{v}) \cdot \mathbf{w} = \alpha(\mathbf{u} \cdot \mathbf{w}) + \beta(\mathbf{v} \cdot \mathbf{w})$, for α, β scalars
- In Euclidean spaces: $\mathbf{u} \cdot \mathbf{u} > 0$ if $\mathbf{u}$ is not zero, and $\mathbf{u} \cdot \mathbf{u} = 0$ if and only if $\mathbf{u}$ is zero

In geometric algebra, the inner product can be applied to any elements of the geometry. Its definition for such arbitrary elements is rather complicated (for instance, it is neither associative nor symmetric, though it is linear), and we defer its definition to Section 2.5, although we will use it before then in some examples to develop a feeling for its meaning.

In our mathematical formulas, we will use '$\cdot$' to represent the inner product. In GABLE, it is the function `inner()`, which takes two arguments:

```
>> inner(2*e1+3*e3, e1)
ans =
      2
```

It is also defined between a scalar α and a vector $\mathbf{u}$, in which case it is defined to by their product: $\alpha \cdot \mathbf{u} = \alpha\mathbf{u}$; however, the converse is zero: $\mathbf{u} \cdot \alpha = 0$.[2] You can also use it on bivectors and trivectors. Try some:

```
>> inner(2,e1)
>> inner(e1,2)
>> inner(e1,e1^e2)
>> inner(e1^e2,e1^e2)
>> inner(e1^e2,-I3)
```

We need to interpret these intriguing results geometrically, and see how we can put this extended inner product to practical use.

[2]The reader who knows geometric algebra can see that we deviate from the commonly used inner product of Hestenes here. We will get back to that.

2.3.2 Interpretation: perpendicularity

For two vectors **u** and **v**, the inner product $\mathbf{u} \cdot \mathbf{v}$ is a scalar. From linear algebra, we know how to interpret this scalar: if two vectors are of unit length, then the inner product is the length of the *perpendicular projection* of each vector on to the other, which is equal to the cosine of the angle between the two vectors. If either vector is not of unit length, then the cosine is scaled by the lengths of the vectors. So if the angle between vectors **u** and **v** is ϕ, then

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}|\ |\mathbf{v}|\ \cos(\phi).$$

Thus the inner product is a measure of *perpendicularity*: if we keep **u** constant and turn **v** to become more and more perpendicular to **u**, the inner product gets smaller, becomes zero when they are precisely perpendicular, and changes sign as **v** moves 'beyond' perpendicular.

The inner product keeps this interpretation of perpendicularity when applied to bivectors, but becomes much more specific geometrically: for instance

$\mathbf{x} \cdot \mathbf{B}$ is a vector in the **B**-plane perpendicular to **x**.

Let us visualize this:

```
>> B = e1^e2;
>> x = e1+e3;
>> xiB = inner(x,B)
>> clf; draw(x,'b'); draw(B,'g'); draw(xiB,'r')
>> GAview([0 90])
```

or you can use `DrawInner()` for visualization:

```
>> clf; DrawInner(x,B);
```

with again the arguments in blue and green on the left, and the result in red on the right.

Note that the result is also perpendicular to **x** and in the **B**-plane if the vector **x** was in the **B**-plane to begin with.

```
>> B = e1^e2;
>> b = e1;
>> biB = inner(b,B)
>> clf; draw(b,'b'); draw(B,'g'); draw(biB,'r')
>> GAview([0 90])
```

So in a sense, the operation '$\cdot\mathbf{B}$', applied to a vector **b** in the **B**-plane (so that $\mathbf{b} \wedge \mathbf{B} = 0$), produces the perpendicular to **b** in that plane, the complement to **b** in the **B**-plane. This generalizes, as we will see later (Section 4.2).

Even with a trivector, the interpretation of the inner product as producing a perpendicular result remains:

```
>> x = e1+e3;
>> clf; draw(x,'b'); draw(inner(x,I3),'r')
>> GAorbiter
```

The inner product $\mathbf{x} \cdot \mathbf{I}_3$ is now the bivector representing the plane perpendicular to **x**. Conversely, the inner product of a bivector with a trivector produces a vector perpendicular to the plane of the bivector:

```
>> B = (e1+e2)^e3;
>> clf; draw(B,'b'); draw(inner(B,I3),'r')
>> GAorbiter
```

You begin to see how conveniently the inner and outer product work together to produce simple expressions for such constructions.

In general, for blades of different grades, if the grade of the first argument is less than the grade of the second argument, then their inner product is a blade whose grade is the difference in grades of the two objects, lies in the subspace of the object of higher grade, and is perpendicular to the object of lower grade. The inner product is thus *grade decreasing*. However, if the first argument has a larger grade than the second, our inner product is zero (because of this grade-reducing property it is also known as a *contraction*, and you cannot contract something bigger onto something smaller).[3]

[3]You should be aware that the more commonly used 'Hestenes inner product' is not a contraction. More about this later.

But what happens if we take the inner product of two blades of the same grade? We already know that the inner product of two vectors yields a scalar. Try taking the inner product of two bivectors:

```
>> inner(e1^e2,e2^e3)
>> inner(e1^e2,e1^e2)
```

In both cases, the result is a scalar; in the first example, the result is 0, while in the second example, it is -1. Likewise, if we take the inner product of the pseudoscalar with itself the result is a scalar. In general, the inner product of two blades of the same grade results in a scalar.

2.3.3 Summary

The inner product is a generalization of the dot product, and may be applied to any two elements of our space. When applied to vectors, it is the familiar dot product. More generally, the inner product is associated with perpendicularity.

2.4 The geometric product

2.4.1 Definition

We have seen how the inner and outer product of two vectors specify aspects of their perpendicularity and parallelness, but neither gives the complete relationship between the two vectors. So it makes sense to combine the two products in a new product. This is the *geometric product*, and it is amazingly powerful.

We denote the geometric product of objects by writing them next to each other leaving the multiplication symbol understood. For *vectors* **u** and **v**, we define:

$$\mathbf{u}\,\mathbf{v} = \mathbf{u}\wedge\mathbf{v} + \mathbf{u}\cdot\mathbf{v}. \tag{1}$$

This is therefore an object of mixed grade: it has a scalar part $\mathbf{u}\cdot\mathbf{v}$ and a bivector part $\mathbf{u}\wedge\mathbf{v}$. This mixed grade is not a problem, as geometric algebra spans a linear space in which such scalar-weighted combinations of blades are perfectly permissible.

Changing the order of **u** and **v** gives:

$$\mathbf{v}\,\mathbf{u} = \mathbf{v}\wedge\mathbf{u} + \mathbf{u}\cdot\mathbf{v} = -\mathbf{u}\wedge\mathbf{v} + \mathbf{u}\cdot\mathbf{v},$$

so the geometric product is neither fully symmetric, nor fully anti-symmetric.

With the geometric product defined, we can use it as the basis of geometric algebra, and view the inner and outer products as secondary, derived constructions. For instance, for vectors we we can retrieve the inner and outer products as its symmetric and anti-symmetric parts, respectively:

$$\begin{aligned}\mathbf{u}\cdot\mathbf{v} &= \tfrac{1}{2}(\mathbf{uv}+\mathbf{vu}) \qquad (2)\\ \mathbf{u}\wedge\mathbf{v} &= \tfrac{1}{2}(\mathbf{uv}-\mathbf{vu}) \qquad (3)\end{aligned}$$

and these formulas can be extended to arbitrary multivectors (see Section 2.5). Although the products *algebraically* have this relationship to each other (with the geometric product being the more fundamental one), yet we will show that *geometrically* it is convenient to think of them as three basic products, each with their own geometric annotations and usage. Inner product and outer product are indeed highly meaningful 'macros' for geometric algebra.

Unfortunately, in Matlab we cannot just omit the multiplication sign, and we have chosen to use '`*`' for the geometric product. Play around with it – but check the outcomes by hand to familiarize yourself with the computations.

```
>> e1*(e1+e2)
>> (e1+e2)*(e1+e3)
>> (e1+e2)*(e1+e2)
>> e1*e2
>> e1*e1
```

The geometric product is extended by linearity and associativity to general multivectors (Section 2.5), which is how we have implemented it. A geometric product of general multivectors may produce multivector with many different grades:

```
>> (1+e1)*(e1*e2 + e1^e2^e3 + inner(e1,e2^e3))
ans =
     e2 + e1^e2 + e2^e3 + I3
```

The geometric interpretation of the geometric product is more difficult than the geometric interpretations of vectors, bivectors, and trivectors. By Equation 1,

```
>> DrawGP(e1,e1+2*e2)
```

results in the scalar $\mathbf{e}_1 \cdot (\mathbf{e}_1 + 2\mathbf{e}_2) = \mathbf{e}_1 \cdot \mathbf{e}_1 = 1$ (indicated in the plot label) and a bivector $\mathbf{e}_1 \wedge (\mathbf{e}_1 + 2\mathbf{e}_2) = \mathbf{e}_1 \wedge \mathbf{e}_2$ (drawn in the figure), and it is hard to understand what that means. We will soon recognize that the geometric product produces a *geometric operator* rather than a *geometric object*, and that therefore we had better visualize it through its effect on objects, rather than by itself.

2.4.2 Invertibility of the geometric product

The inner product and outer product each specify the relationships between vectors incompletely, so they can *not* be inverted (e.g., knowing the value of the inner product of an unknown vector $\mathbf{x}$ with a known vector $\mathbf{u}$ does not tell you what $\mathbf{x}$ is). However, *the geometric product is invertible.* This is extremely powerful in computations.

Still, not all multivectors have inverses in general geometric algebras. Fortunately, in the geometric algebra of Euclidean space, subspaces (represented as *blades*, i.e., multivectors of a single grade) do. Thus, for each blade $\mathbf{A} \neq 0$ in Euclidean space, we can find $\mathbf{A}^{-1}$ such that $\mathbf{A}\mathbf{A}^{-1} = 1$.

Let us first take a linear subspace, characterized by a vector $\mathbf{v}$. (Why does a vector characterize a linear subspace? Because $\mathbf{x} \wedge \mathbf{v} = 0$ characterizes all vectors $\mathbf{x}$ in this subspace.) What is its inverse? Think about this, then ask GABLE:

```
>> v = 2*e1;
>> iv = inverse(v)
>> v*iv
>> iv*v
```

So, as you thought, the inverse of a vector is parallel to the vector, but differs by a scalar factor:

$$\mathbf{v}^{-1} = \frac{\mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}.$$

This is easily verified:

$$\mathbf{v}(\mathbf{v}/(\mathbf{v} \cdot \mathbf{v})) = (\mathbf{v}\mathbf{v})/(\mathbf{v} \cdot \mathbf{v}) = (\mathbf{v} \cdot \mathbf{v} + \mathbf{v} \wedge \mathbf{v})/(\mathbf{v} \cdot \mathbf{v}) = (\mathbf{v} \cdot \mathbf{v} + 0)/(\mathbf{v} \cdot \mathbf{v}) = 1.$$

Now consider a two-dimensional subspace, characterized by a bivector $\mathbf{B}$. For instance, what is the inverse of $2\mathbf{e}_1 \wedge \mathbf{e}_2$, where the basis $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ is orthonormal? In such a basis we have:

$$\mathbf{e}_1\,\mathbf{e}_1 = \mathbf{e}_1 \cdot \mathbf{e}_1 = 1, \quad \text{etc.} \quad \text{and} \quad \mathbf{e}_1\,\mathbf{e}_2 = \mathbf{e}_1 \wedge \mathbf{e}_2, \quad \text{etc.} \tag{4}$$

With this, we observe that

$$(\mathbf{e}_1 \wedge \mathbf{e}_2)\,(\mathbf{e}_1 \wedge \mathbf{e}_2) = (\mathbf{e}_1\mathbf{e}_2)(\mathbf{e}_1\mathbf{e}_2) = \mathbf{e}_1(\mathbf{e}_2\mathbf{e}_1)\mathbf{e}_2 = -(\mathbf{e}_1\mathbf{e}_1)(\mathbf{e}_2\mathbf{e}_2) = -1, \tag{5}$$

so, in the sense of the geometric product: *the square of a bivector is negative.* Then the inverse is simple to determine: $(2\mathbf{e}_1 \wedge \mathbf{e}_2)^{-1} = -\frac{1}{2}(\mathbf{e}_1 \wedge \mathbf{e}_2) = \frac{1}{2}\mathbf{e}_2 \wedge \mathbf{e}_1$. In general, in $\mathcal{C}\ell_{3,0}$

$$\mathbf{B}^{-1} = \frac{\mathbf{B}}{\mathbf{B} \cdot \mathbf{B}}.$$

The inverse of a pseudoscalar $\alpha\mathbf{I}_3$ is also easy. Observe that

$$\begin{aligned}
\mathbf{I}_3\mathbf{I}_3 &= (\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3)(\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3) \\
&= \mathbf{e}_1\mathbf{e}_2\mathbf{e}_3\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3 = -\mathbf{e}_2\mathbf{e}_1\mathbf{e}_3\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3 \\
&= \mathbf{e}_2\mathbf{e}_3\mathbf{e}_1\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3 = \mathbf{e}_2\mathbf{e}_3\mathbf{e}_2\mathbf{e}_3 \\
&= -\mathbf{e}_3\mathbf{e}_2\mathbf{e}_2\mathbf{e}_3 = -\mathbf{e}_3\mathbf{e}_3 = -1
\end{aligned}$$

Thus, in our algebra the inverse of $\alpha\mathbf{I}_3$ is:

$$(\alpha\mathbf{I}_3)^{-1} = -\mathbf{I}_3/\alpha.$$

In GABLE, use `inverse()` to compute the inverse of a geometric object. As a short-hand for `B*inverse(A)`, you may write `B/A`. Note that the geometric product is *not* in general commutative, so we would write `inverse(A)*B` as `(1/A)*B`, which is rarely equal to `B/A`.

2.4.3 Duality

The *dual* of an element $\mathbf{A}$ of our geometric algebra is defined to be

$$\text{dual}\mathbf{A} \equiv \mathbf{A}/\mathbf{I}_3 = -\mathbf{A}\mathbf{I}_3, \tag{6}$$

and in GABLE the command `dual` returns the dual of an object. The dual of a blade representing a subspace is a blade representing the *orthogonal complement* of that subspace, i.e., the space of all vectors perpendicular to it. This is a common construction, and it is great to have it in such a simple form: just divide by $\mathbf{I}_3$.

For example, type the following:

```
>> clf
>> draw(e1)
>> draw(dual(e1))
```

The blue arrow represents `e1`; the green circle represents the dual of `e1`, which is `-e2^e3` as shown by the following derivation:

$$\begin{aligned}
\text{dual}(\mathbf{e}_1) &= \mathbf{e}_1\mathbf{I}_3^{-1} = \mathbf{e}_1(\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3)^{-1} \\
&= \mathbf{e}_1(\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3)^{-1} = -\mathbf{e}_1\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3 \\
&= -\mathbf{e}_2\mathbf{e}_3 = -\mathbf{e}_2 \wedge \mathbf{e}_3
\end{aligned}$$

The construction is more striking for more arbitrary vectors, of course (see exercises).

Note that for a blade $\mathbf{U}$ we have grade (dual($\mathbf{U}$)) $= 3 -$ grade ($\mathbf{U}$), so that the dual of a scalar is a pseudoscalar and vice versa: dual(1) $= -\mathbf{I}_3$ and dual($\mathbf{I}_3$) $= 1$ (this is true in any space and partly explains the name 'pseudoscalar' for the n-dimensional volume element).

As a consequence of this rule on grades, the dual relationship between vectors and bivectors is *only* valid in 3-dimensional space. In 3-space, we may characterize a plane (which is really a bivector) dually by a vector; that vector is commonly called the *normal vector* of the plane. We now see that it is the dual of the bivector. Indeed, both of the following two equations characterize the same plane $\mathbf{B}$:

$$\mathbf{x} \wedge \mathbf{B} = 0 \quad \text{and} \quad \mathbf{x} \cdot \text{dual}(\mathbf{B}) = 0.$$

We recognize the latter as the 'normal equation' of the plane $\mathbf{B}$, the inner product of a vector $\mathbf{x}$ with the normal vector $\mathbf{n} \equiv \text{dual}(\mathbf{B}) = \mathbf{B}/\mathbf{I}_3$.

This is an example of a duality relationship between the outer product and inner product. Since the outer product produces an element of geometric algebra, we can take its dual. One can then prove (nice exercise, try it for vectors)

$$\text{dual}(\mathbf{u} \wedge \mathbf{v}) = \mathbf{u} \cdot \text{dual}(\mathbf{v}) \quad \text{and} \quad \text{dual}(\mathbf{u} \cdot \mathbf{v}) = \mathbf{u} \wedge \text{dual}(\mathbf{v}) \tag{7}$$

for any multivectors $\mathbf{u}$ and $\mathbf{v}$ from the geometric algebra of the space with the pseudoscalar $\mathbf{I}_3$ used to define the dual.

This is a good moment to explain how the 3-dimensional *cross product* of vectors fits into geometric algebra. The cross product obeys

$$\mathbf{u} \times \mathbf{v} = \text{dual}(\mathbf{u} \wedge \mathbf{v}).$$

You can see this with the following GABLE commands:

```
>> a = e1+0.5*e2;
>> b = e1+e2+e3;
>> B = a^b;
>> draw(a);draw(b);
>> draw(B,'g');
>> draw(dual(B),'r');
>> GAorbiter
```

Here we see that the red vector is perpendicular to both of the blue vectors. To check that the dual matches the cross-product, print the dual of `B` and then use the Matlab `cross` command to compute the cross-product of the two vectors.

So we could define the cross product using the above equation. However, we will *not* do so, for two reasons. Firstly, the cross product is too particular for 3-dimensional space; in no other space is the dual of a 'span of vectors' a vector of that space. And secondly, its only use is to characterize planes and rotations. Geometric algebra offers a much more convenient way to characterize those, directly through bivectors. We have seen this for planes and we will see it for rotations soon. Since these bivector characterizations are valid in arbitrary dimensions, we prefer them to any specific, 3D-only construction. So: *we will not need the cross product.*

Exercises

1. Determine the subspace perpendicular to the vector $\mathbf{e}_1 + 0.2\,\mathbf{e}_3$.
2. Determine the subspace perpendicular to the plane spanned by the vectors $\mathbf{e}_1 + 0.3\,\mathbf{e}_3$ and $\mathbf{e}_2 + 0.5\,\mathbf{e}_3$, in one line of GABLE.
3. Prove Equation 7.
4. We used the geometric product to define the dual. We can also make the dual using the inner product (thus directly conveying the intuition of 'orthogonal complement'). Give such a formulation of the dual in a way that is equivalent to the geometric product formulation of the dual and show this equivalence.

2.4.4 Summary

The geometric product is a third product of our geometric algebra. Unlike the inner and outer products, the geometric product is invertible, which is useful when doing algebraic manipulations. In Section 3, we will see geometric interpretations of the geometric product.

2.5 Extension of the products to general multivectors

We have stated that the inner, outer, and geometric products can be generalized to arbitrary multivectors. In this section, we will first show how to extend the definition of the geometric product to general multivectors. Once we have that, it is easy to extend the inner and outer products.

For general objects of geometric algebra, the geometric product can be defined as follows (this is not the only way, but it is the most easy to understand). In the n-dimensional vector space considered, introduce an orthogonal basis $\{\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_n\}$. Use the outer product to extend this to a basis for the whole geometric algebra (the one containing $\mathbf{e}_1 \wedge \mathbf{e}_2$, etcetera). Any multivector can be written as a weighted sum of basis elements on this extended basis. The geometric product is defined to be *linear* and *associative* in its arguments, and *distributive over* $+$, so it is sufficient to defined what the result is of combining two arbitrary elements of the extended basis. We first observe that the desired compatibility with Equation 1 combined with the orthogonality of the basis vectors leads to

$$\mathbf{e}_i\mathbf{e}_j = -\mathbf{e}_j\mathbf{e}_i \quad \text{if} \quad i \neq j \tag{8}$$

because on the orthogonal basis, effectively $\mathbf{e}_i\mathbf{e}_j$ equals $\mathbf{e}_i \wedge \mathbf{e}_j$ if $i \neq j$. If $i = j$, Equation 1 gives the scalar result $\mathbf{e}_i \cdot \mathbf{e}_i$, which in our Euclidean space equals 1.[4] So we have

$$\mathbf{e}_i\mathbf{e}_i = 1. \tag{9}$$

This is now enough to define the geometric product of any elements in the geometric algebra. For instance $(2+\mathbf{e}_1\wedge\mathbf{e}_2)(\mathbf{e}_1+\mathbf{e}_2\wedge\mathbf{e}_3)$ is expanded by distributivity over $+$ to $2\mathbf{e}_1+(\mathbf{e}_1\wedge\mathbf{e}_2)\mathbf{e}_1+2(\mathbf{e}_2\wedge\mathbf{e}_3)+(\mathbf{e}_1\wedge\mathbf{e}_2)(\mathbf{e}_2\wedge\mathbf{e}_3)$. The term $(\mathbf{e}_1\wedge\mathbf{e}_2)\mathbf{e}_1$ in this equals $(\mathbf{e}_1\mathbf{e}_2)\mathbf{e}_1$, and by associativity this equals $\mathbf{e}_1\mathbf{e}_2\mathbf{e}_1$. We apply Equation 8 to get $-\mathbf{e}_1\mathbf{e}_1\mathbf{e}_2$, and then Equation 9 to get $-\mathbf{e}_2$. The other terms are computed in a similar way. Note that this definition is heavily based on the introduction of an orthogonal basis (which is somewhat inelegant); other definitions manage to avoid that. You may also think that all these expansions make the geometric product an expensive operation. But the above was just to show how those minimal definitions actually define the outcome mathematically; a more practical computation scheme using matrices on the extended basis is what we used to make GABLE (such details may be found in [12, 13]).

Now that we have defined the general geometric product, it is easy to generalize both the inner and outer products. Both products are linear in their arguments, and so are sufficiently specified when we say what they do on blades. For instance, if we would want to know the

[4] To get a general geometric algebra, of a space with a quadratic form ('metric') Q, this is set to some specified scalar $Q(\mathbf{e}_i)$, usually taken to be $+1$ or -1. The sign of $Q(\mathbf{e}_i)$ is called the *signature* of $\mathbf{e}_i$.

outcome of $(\mathbf{A}_1 + \mathbf{A}_2) \cdot (\mathbf{B}_1 + \mathbf{B}_2)$ (where the index denotes the grade of the blades involved), then this can be written out as $\mathbf{A}_1 \cdot \mathbf{B}_1 + \mathbf{A}_1 \cdot \mathbf{B}_2 + \mathbf{A}_2 \cdot \mathbf{B}_1 + \mathbf{A}_2 \cdot \mathbf{B}_2$.

For a blade $\mathbf{U}$ of grade r, and a blade $\mathbf{V}$ of grade s, the definitions for inner and outer products are:

$$\mathbf{U} \cdot \mathbf{V} = \text{grade}(\mathbf{UV}, s - r)$$
$$\mathbf{U} \wedge \mathbf{V} = \text{grade}(\mathbf{UV}, s + r).$$

Since no element of geometric algebra has a negative grade, the inner product is only non-zero if $s \geq r$.[5] Note that the inner product lowers the grade, and the outer product increases the grade.

For a vector $\mathbf{u}$ and an s-blade $\mathbf{V}$, these formulas can be shown to produce:

$$\mathbf{u} \wedge \mathbf{V} = \tfrac{1}{2}(\mathbf{uV} + \widehat{\mathbf{V}}\mathbf{u}) \tag{10}$$
$$\mathbf{u} \cdot \mathbf{V} = \tfrac{1}{2}(\mathbf{uV} - \widehat{\mathbf{V}}\mathbf{u}). \tag{11}$$

where we used $\widehat{\mathbf{V}}$ as a shorthand for $(-1)^s\mathbf{V}$ (it is sometimes called the *grade involution*). Compare this to equations (3) and (2).

Beware: it is *not* generally true that $\mathbf{UV} = \mathbf{U} \cdot \mathbf{V} + \mathbf{U} \wedge \mathbf{V}$; that is only so if $\mathbf{U}$ is a *vector*. For instance, compute the geometric product of two specific bivectors:

```
>> (e1^e2)*((e1+e2)^e3)
ans =
     -1*e2^e3 + -1*e3^e1
>> inner((e1^e2),((e1+e2)^e3))
ans =
     0
>> (e1^e2)^((e1+e2)^e3)
ans =
     0
```

In general, the geometric product of an r-vector and an s-vector contains vectors of grade $|r-s|, |r-s|+2, \cdots r+s-2, r+s$; the inner and outer product specify only two terms of this sequence, and are therefore only a partial representation of the geometric product (which contains *all* geometric relationships between its arguments). For objects other than vectors, there is much more to geometric algebra than just perpendicularity (inner product) and spanning (outer product), but in this tutorial we focus on those.

There are some useful formulas permitting the computation of the inner product of multi-vectors made using the geometric product or the outer product. We state them without proof, for vectors $\mathbf{u}$ and pure blades $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ (the general case then follows by linearity).

$$\mathbf{u} \cdot (\mathbf{VW}) = (\mathbf{u} \cdot \mathbf{V})\mathbf{W} + \widehat{\mathbf{V}}(\mathbf{u} \cdot \mathbf{W}) \tag{12}$$
$$\mathbf{u} \cdot (\mathbf{V} \wedge \mathbf{W}) = (\mathbf{u} \cdot \mathbf{V}) \wedge \mathbf{W} + \widehat{\mathbf{V}} \wedge (\mathbf{u} \cdot \mathbf{W}) \tag{13}$$
$$(\mathbf{U} \wedge \mathbf{V}) \cdot \mathbf{W} = \mathbf{U} \cdot (\mathbf{V} \cdot \mathbf{W}) \tag{14}$$

3 Geometry

In this section we will show how the products of geometric algebra can be used to perform many geometrical tasks.

3.1 Projection, rejection

Given a subspace and a vector, one operation we commonly need to perform is to find the part of the vector that lies in the subspace and the part of the vector that lies outside the subspace. These operations are called *projection* and *rejection* respectively. Both are easy to perform with geometric algebra.

[5] In this the inner product we use as a default in GABLE deviates from the most commonly used inner product in geometric algebra, as defined by Hestenes, which has $|s-r|$ rather than $(s-r)$. Our product has a more direct geometric interpretation, which we will need in the later chapters.

Let us begin with a vector $\mathbf{v}$ and the desire to write it as $\mathbf{v}_\perp + \mathbf{v}_\|$ relative to a subspace characterized by a blade $\mathbf{M}$, where $\mathbf{v}_\perp$ is the component of $\mathbf{v}$ perpendicular to $\mathbf{M}$, and $\mathbf{v}_\|$ the parallel component. Therefore $\mathbf{v}_\perp$ and $\mathbf{v}_\|$ need to satisfy

$$\mathbf{v}_\perp \cdot \mathbf{M} = 0 \ \text{ and } \ \mathbf{v}_\| \wedge \mathbf{M} = 0.$$

Thus

$$\begin{aligned}\mathbf{v}_\perp\mathbf{M} &= \mathbf{v}_\perp \cdot \mathbf{M} + \mathbf{v}_\perp \wedge \mathbf{M}\\ &= \mathbf{v}_\perp \wedge \mathbf{M}\\ &= \mathbf{v}_\perp \wedge \mathbf{M} + \mathbf{v}_\| \wedge \mathbf{M}\\ &= \mathbf{v} \wedge \mathbf{M}.\end{aligned}$$

But we can divide by the blade $\mathbf{M}$, on both sides, so we obtain:

$$\mathbf{v}_\perp = (\mathbf{v} \wedge \mathbf{M})/\mathbf{M} \tag{15}$$

A similar derivation shows that

$$\mathbf{v}_\| = (\mathbf{v} \cdot \mathbf{M})/\mathbf{M}. \tag{16}$$

These are the general projection and rejection formulas for a vector $\mathbf{v}$ relative to *any* subspace with invertible blade $\mathbf{M}$, in any number of dimensions. Powerful stuff!

It is important to visualize what is going on. Take $\mathbf{M}$ to be a 2-blade, determining a plane. Then $\mathbf{v}\wedge\mathbf{M}$ is a volume spanned by $\mathbf{v}$ with that plane. It is a 'reshapable' volume: any vector $\mathbf{v}$ that has its endpoint on the plane parallel to $\mathbf{M}$, through $\mathbf{v}$'s endpoint, spans the same trivector. The division by $\mathbf{M}$ in the formula for the projection demands the factoring of this volume into a component $\mathbf{M}$, and therefore returns what is left: the unique vector perpendicular to $\mathbf{M}$ that spans the volume. This is a general property:

Dividing a space $\mathbf{B}$ *by a subspace* $\mathbf{A}$ *produces the orthogonal complement to* $\mathbf{A}$ *in* $\mathbf{B}$.

If $\mathbf{A}$ is *not* a proper subspace, other things happen that we'll cover later.

These relationships can be seen in GABLE. To begin, clear the graphics screen (`clf`) and draw a bivector. For this demonstration, we will use `DrawBivector` to draw the graphical representation of a bivector. Next, draw a vector that lies outside the plane of this bivector:

```
>> clf; B = DrawBivector(e1,e1+e2)
>> v = 1.5*e1+e2/3+e3;
>> draw(v);
```

To get a better feel for the 3D relationships, use `GAorbiter` to rotate around the scene a bit.

We can now type our formulas for the perpendicular and parallel parts of `v` directly into GABLE and draw the resulting perpendicular and parallel components of `v` relative to `B`:

```
>> %... needs v,B
>> vpar = inner(v,B)/B;
>> draw(vpar);
>> vperp = (v^B)/B;
>> draw(vperp,'r')
```

We stated that this computation works for any subspace $\mathbf{B}$. In particular, we can set `B` to be a vector, and the same computations for `vpar` and `vperp` work. Try this!

As we will see later (but you could try it now), you may project a bivector $\mathbf{A}$ onto a bivector $\mathbf{B}$ using the 'same' formula: $\mathbf{A}_\| = (\mathbf{A}\cdot\mathbf{B})/\mathbf{B}$. However, the rejection of the bivector is now *not* obtained by $\mathbf{A}_\perp = (\mathbf{A}\wedge\mathbf{B})/\mathbf{B}$, (which is zero since quadvectors do not exist in 3D), but simply by $\mathbf{A}_\perp = \mathbf{A} - \mathbf{A}_\|$ (a formula that also works in the previous case where $\mathbf{A}$ is a vector). More about the relationships of subspaces in Section 4.

Exercises

1. Redo the example of projection and rejection using the same `v` as above, but with `B=e1+e2`. You will need to use `GAorbiter` to see the perpendicular relationship, although it can also be tested non-graphically using `inner()`.
2. Let $\mathbf{v} = -3\mathbf{e}_1 - 2\mathbf{e}_2$ and let $\mathbf{B} = 2\mathbf{e}_2 \wedge (\mathbf{e}_3 + 3\mathbf{e}_1)$. Compute the projection and rejection of $\mathbf{v}$ by $\mathbf{B}$. Draw $\mathbf{v}$, $\mathbf{B}$, and this projection and rejection. Try this exercise once using `DrawBivector` to draw $\mathbf{B}$ and a second time using `draw` to draw $\mathbf{B}$.

3. With the same $\mathbf{v}$ and $\mathbf{B}$ as the previous exercise, study how the rejection formula works: first draw $\mathbf{v} \wedge \mathbf{B}$ using `DrawTrivector`; as decomposition use $\mathbf{v}$, $\mathbf{v} \cdot \mathbf{B}$ and the projection $\mathrm{P}_((\mathbf{v}), \mathbf{B})$. Then draw that trivector again in a decomposition that uses the rejection.
4. Prove the formula for projection: $\mathbf{v}_\| = (\mathbf{v} \cdot \mathbf{M})/\mathbf{M}$.
5. (Not easy!) Using Equations 12, 13 and 14, show that

$$((\mathbf{a} \wedge \mathbf{b}) \cdot \mathbf{M})/\mathbf{M} = \mathbf{a}_\| \wedge \mathbf{b}_\|$$

(still a useful exercise!). This gives two ways to compute the projection of a bivector $\mathbf{a} \wedge \mathbf{b}$ relative to a blade $\mathbf{M}$. Which do you prefer and why?

3.2 Orthogonalization

Geometric algebra does *not* require the representation of its elements in terms of a particular basis of vectors. Therefore the specific treatment of issues like orthogonalization are much less necessary. Yet it is sometimes convenient to have an orthogonal basis, and they are simple to construct using our products.

Suppose we have a set of three vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, and would like to form them into an orthogonal basis. We arbitrarily keep $\mathbf{u}$ as one of the vectors of the perpendicularized frame, which will have vectors denoted by primes:

$$\mathbf{u}' \equiv \mathbf{u}.$$

Then we form the rejection of $\mathbf{v}$ by $\mathbf{u}'$, which is perpendicular to $\mathbf{u}'$:

$$\mathbf{v}' \equiv (\mathbf{v} \wedge \mathbf{u}')/\mathbf{u}'$$

Now we take the rejection of $\mathbf{w}$ by $\mathbf{u}' \wedge \mathbf{v}'$, which is perpendicular to both $\mathbf{u}'$ and $\mathbf{v}'$:

$$\mathbf{w}' \equiv (\mathbf{w} \wedge \mathbf{u}' \wedge \mathbf{v}')/(\mathbf{u}' \wedge \mathbf{v}')$$

and we are done. (This is the Gram-Schmidt orthogonalization procedure, rewritten in geometric algebra.) Here's an example (no need to type it, use `GAblock(1)`, see section 1.2!):

```
>> %GAblock(1)
>> % ORTHOGONALIZATION
>> clf;
>> u = e1+e2;
>> v = 0.3*e1 + 0.6*e2 - 0.8*e3;
>> w = e1 -0.2*e2 + 0.5*e3;
>> up = u;
>> vp = (v^up)/up;
>> wp = (w^up^vp)/(up^vp);
>> draw(u); draw(v); draw(w);                 %% The original vectors ...
>> draw(up,'r'); draw(vp,'r'); draw(wp,'r');  %% ... and orthognalized
>> GAorbiter
```

You might want to draw the duals to show the perpendicularity of the resulting basis more clearly (see Exercise 1).

Note that in this construction, $\mathbf{v}' \wedge \mathbf{u}' = \mathbf{v}'\mathbf{u}' = \mathbf{v} \wedge \mathbf{u}$, and $\mathbf{w}' \wedge \mathbf{v}' \wedge \mathbf{u}' = \mathbf{w}'(\mathbf{v}'\mathbf{u}') = \mathbf{w} \wedge \mathbf{v} \wedge \mathbf{u}$, so that it preserves the trivector spanned by the basis, in magnitude and orientation. Check this in GABLE:

```
>> u^v^w == up^vp^wp
ans =
     1
```

As an exercise, you might want to give the algorithm for n-dimensional orthogonalization.

Exercises

1. Convince yourself that `up`, `vp` and `wp` are orthogonal, using graphics routines to explore their construction. You might for instance draw `dual(up)`, and in that plane study `vp` and `wp`.

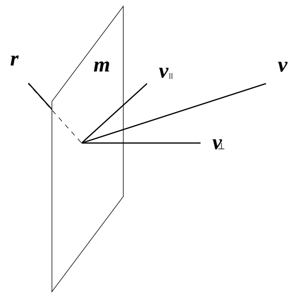

Figure 3: Reflecting v through m.

3.3 Reflection

Suppose we wish to reflect a vector $\mathbf{v}$ through some subspace (a vector, a plane, whatever). In a geometric algebra, this subspace is naturally described by a blade, so we will look at reflecting $\mathbf{v}$ through a unit blade $\mathbf{M}$. If we write $\mathbf{v}$ as

$$\mathbf{v} = \mathbf{v}_\perp + \mathbf{v}_\|,$$

where $\mathbf{v}_\perp$ is the part of $\mathbf{v}$ perpendicular to $\mathbf{M}$ and $\mathbf{v}_\|$ is the part parallel to $\mathbf{M}$ (we derived formulas for both in the Section 3.1), then $\mathbf{r}$, the reflection of $\mathbf{v}$ through $\mathbf{M}$, is given by

$$\mathbf{r} = \mathbf{v}_\| - \mathbf{v}_\perp.$$

Using our formulas for the parallel and perpendicular parts of $\mathbf{v}$ relative to $\mathbf{M}$, we see that

$$\begin{aligned} \mathbf{r} &= \mathbf{v}_\| - \mathbf{v}_\perp \\ &= (\mathbf{v}\cdot\mathbf{M})\mathbf{M}^{-1} - (\mathbf{v}\wedge\mathbf{M})\mathbf{M}^{-1} \\ &= \tfrac{1}{2}(\mathbf{v}\mathbf{M} - \widehat{\mathbf{M}}\mathbf{v})\mathbf{M}^{-1} - \tfrac{1}{2}(\mathbf{v}\mathbf{M} + \widehat{\mathbf{M}}\mathbf{v})\mathbf{M}^{-1} \\ &= -\widehat{\mathbf{M}}\mathbf{v}\mathbf{M}^{-1}. \end{aligned}$$

This is an interestingly simple expression for such arbitrary reflections.

Let us see what the formula yields for specific choices of the blade we reflect in. In 3D, there are four possibilities for the blades.

- *scalar:* $M = 1$
  The scalar, viewed as a subspace, is a point at the origin (see Section 2.2.5). And indeed, our reflection formula yields $\mathbf{v} \;\rightarrow\; -\mathbf{v}$, which is clearly a point reflection in the origin.
- *vector:* $\mathbf{M} = \mathbf{u}$
  Let the blade $\mathbf{M}$ characterize the reflecting subspace be a unit vector $\mathbf{u}$; then the reflection is relative to a line in direction $\mathbf{u}$. This gives

  $$\mathbf{r} = \mathbf{u}\mathbf{v}\mathbf{u}.$$

  An attractive formula for a basic process!
- *bivector* $\mathbf{M} = \mathbf{B}$
  For a plane in 3D characterized by a unit bivector $\mathbf{B}$ (so that $\mathbf{B}^2 = -1$, and therefore $\mathbf{B}^{-1} = -\mathbf{B}$), we obtain

  $$\mathbf{r} = \mathbf{B}\mathbf{v}\mathbf{B}.$$

  We could also write this in terms of the dual to the plane, i.e., the normal vector $\mathbf{n}$ defined as $\mathbf{n} = \mathbf{B}\mathbf{I}_3^{-1}$:

  $$\mathbf{r} = \mathbf{B}\mathbf{v}\mathbf{B} = \mathbf{n}\mathbf{I}_3\mathbf{v}\mathbf{n}\mathbf{I}_3 = \mathbf{n}\mathbf{I}_3^2\mathbf{v}\mathbf{n} = -\mathbf{n}\mathbf{v}\mathbf{n}.$$

  (It is a good exercise to prove $\mathbf{r} = -\mathbf{n}\mathbf{v}\mathbf{n}$ for reflection in a hyperplane dual to $\mathbf{n}$ in arbitrary-dimensional space.)

  You can see this in GABLE with the following sequence of commands:

```
>> v = e1+2*e3;
>> M = e1^e2;
>> draw(v)
>> draw(M,'g')
>> draw(M*v*M,'r')
```

- *volume* $\mathbf{M} = \mathbf{I}_3$
  Since $\widehat{\mathbf{I}}_3 = -\mathbf{I}_3$ and $\mathbf{I}_3$ commutes with $\mathbf{v}$, this gives $\mathbf{v} \to \mathbf{v}$. Reflection in the containing space is the identity, since $\mathbf{v}_\perp = 0$.

To develop the formula for the reflection of an arbitrary blade of grade k in the blade $\mathbf{M}$ of grade m, first see what happens for the geometric product of the reflection of k vectors:

$$\begin{aligned}(-\widehat{\mathbf{M}}\mathbf{u}_1\mathbf{M}^{-1})(-\widehat{\mathbf{M}}\mathbf{u}_2\mathbf{M}^{-1})\cdots(-\widehat{\mathbf{M}}\mathbf{u}_k\mathbf{M}^{-1}) =& \\ = \; & (-1)^k\widehat{\mathbf{M}}\mathbf{u}_1\mathbf{M}^{-1}\widehat{\mathbf{M}}\mathbf{u}_2\mathbf{M}^{-1}\cdots\widehat{\mathbf{M}}\mathbf{u}_m\mathbf{M}^{-1} \\ = \; & (-1)^{k+km}\mathbf{M}\mathbf{u}_1\mathbf{M}^{-1}\mathbf{M}\mathbf{u}_2\mathbf{M}^{-1}\cdots\mathbf{M}\mathbf{u}_k\mathbf{M}^{-1} \\ = \; & (-1)^{k(m+1)}\mathbf{M}\mathbf{u}_1\mathbf{u}_2\cdots\mathbf{u}_k\mathbf{M}^{-1}.\end{aligned}$$

Therefore:

$$\begin{aligned}\mathbf{X}\text{ even grade}: \quad & \mathbf{X} \to \mathbf{M}\mathbf{X}\mathbf{M}^{-1} \\ \mathbf{X}\text{ odd grade}: \quad & \mathbf{X} \to -\widehat{\mathbf{M}}\mathbf{X}\mathbf{M}^{-1}\end{aligned}$$

A bit complicated; but that is how reflections are for such subspaces.

By the way, you see from this what happens if you reflect a plane characterized by a bivector $\mathbf{B}$, relative to the origin: it is preserved. However, its normal would reflect, and that is undesirable. In the past, when people could only characterize planes by normal vectors, they therefore had to make two kinds of vectors: position vectors which do reflect in the origin, and 'axial' vectors which don't, and realize that a normal vector is an axial vector. Using bivectors directly to characterize planes, we have no need of such a strange distinction: the algebra and its semantics simplifies by admitting objects of higher grade!

Exercises

1. Reflect the bivector `B = (e1^e2+2/3*e2^e3+e3^e1/3)` through the plane spanned by the bivector `M = e1^e2`. Note carefully what happens to the orientation of the bivector. Is that what you expected?
2. Reflect the pseudoscalar `I3` through a bivector. Note the orientation.
3. Do orientation and magnitude of the reflecting blade $\mathbf{M}$ matter to the reflection result?

3.4 Rotations

3.4.1 Rotations in a plane

Consider a vector $\mathbf{a}$, and a vector $\mathbf{b}$ that is 'a rotated version of $\mathbf{a}$'. We can try to construct $\mathbf{b}$ as the multiplication of $\mathbf{a}$ with some element $\mathbf{R}$ of our algebra:

$$\mathbf{b} = \mathbf{R}\mathbf{a},$$

and since the geometric product with $\mathbf{a}$ is invertible we find simply that

$$\mathbf{R} = \mathbf{b}/\mathbf{a} = \frac{\mathbf{b}\mathbf{a}}{\mathbf{a}\cdot\mathbf{a}}.$$

This is a bit too general to describe a rotation, since we have nowhere demanded explicitly that $\mathbf{a}$ and $\mathbf{b}$ should have the same norm. Therefore $\mathbf{ba}$ contains both a rotation and a dilation ('stretch') of the vector $\mathbf{a}$. This is then a possible *interpretation for the geometric product:* $\mathbf{ba}$ *is the operator that maps* $\mathbf{a}^{-1}$ *to* $\mathbf{b}$. (Or, alternatively, $\mathbf{b}/\mathbf{a}$ is the operator that transforms $\mathbf{a}$ into $\mathbf{b}$.)

For the pure rotation we wished to consider, the length of the rotated vector (and equivalently, the norm of the vector) does not change. To create an $\mathbf{R}$ that does not depend on the length of the vectors we used to construct it, we should construct $\mathbf{R}$ using vectors of the same norm in the direction of $\mathbf{a}$ and $\mathbf{b}$; for instance, both unit vectors. In Euclidean space the inverse of a unit vector is the vector itself. So then we get: *a rotation operator is the geometric product of two unit vectors* (and vice versa).

Try this in GABLE using the following sequence of commands:

```
>> a = e1;
>> b = unit(e1+e2);
>> R = b/a;
>> clf; draw(a); draw(b);
```

At this point, we see the vectors `a` and `b` drawn in blue. By construction, they are separated by a 45 degree angle. If you now type:

```
>> %... needs a,R
>> draw(R*a,'r');
```

we see a red vector drawn over-top `b`, since this is `a` rotated to become `b`. Rotating this vector a second time and looking from above

```
>> %... needs a,R
>> draw(R*R*a,'g');
>> GAview([0 90]);
```

we see that we have rotated `a` by the angle between `a` and `b` (45 degrees) twice, giving rotation of `a` by 90 degrees (drawn in green).

You can use this rotation to rotate any vector in the $\mathbf{a} \wedge \mathbf{b}$-plane; for instance, if you wish to rotate $\mathbf{c} = 2\mathbf{a} + \mathbf{b}$, then you are looking for a vector $\mathbf{d}$ such that $\mathbf{d}$ is to $\mathbf{c}$ as $\mathbf{b}$ is to $\mathbf{a}$. In formula: $\mathbf{d}/\mathbf{c} = \mathbf{b}/\mathbf{a}$, and therefore $\mathbf{d} = (\mathbf{b}/\mathbf{a})\mathbf{c} = \mathbf{R}\mathbf{c}$. Let us try that:

```
>> %... needs a,b
>> c = 2*a + b;
>> draw(c,'m');
>> draw(R*c,'m');
```

Writing things in their components of fixed grade, the expression for $\mathbf{R}$ is (taking $\mathbf{a}$ and $\mathbf{b}$ unit vectors)

$$\mathbf{R} = \mathbf{b}\mathbf{a} = \mathbf{b} \cdot \mathbf{a} + \mathbf{b} \wedge \mathbf{a} = \cos\phi - \mathbf{i}\sin\phi,$$

where $\mathbf{i}$ is the bivector denoting the oriented plane containing $\mathbf{a}$ and $\mathbf{b}$, oriented *from* $\mathbf{a}$ *to* $\mathbf{b}$, and ϕ is the angle between them in that oriented plane, so *from* $\mathbf{a}$ *to* $\mathbf{b}$. Note that if we would orient the plane from $\mathbf{b}$ to $\mathbf{a}$, then $\mathbf{i}$ would change sign, and $\mathbf{R}$ would only be the same if ϕ changes sign as well. So the orientation of $\mathbf{i}$ specifies how to measure the angle.

If you have had complex numbers or Taylor series in your math courses, you may realize that we can write the above as an exponential:

$$\cos\phi - \mathbf{i}\sin\phi = e^{-\mathbf{i}\phi}$$

This is based on $\mathbf{i}^2 = -1$, which is correct since $\mathbf{i}$ is a unit bivector in $\mathcal{C}\ell_{3,0}$ (Equation 5). If you have not had those subjects, just consider this as a definition of a convenient shorthand notation in terms of 'abstract' exponentials. In fact, there are several equivalent ways of writing the relationship between $\mathbf{a}$ and $\mathbf{b}$, and hence the rotation over the 'bivector angle' $\mathbf{i}\phi$:

$$\mathbf{b} = e^{-\mathbf{i}\phi}\mathbf{a} = \mathbf{a}e^{\mathbf{i}\phi}.$$

The sign-change in the exponent is due to the non-commutative properties of the geometric product. So $\mathbf{i}$ is not really a complex number: then the order would not have mattered.

As we showed before when rotating the vector $\mathbf{c} = 2\mathbf{a} + \mathbf{b}$, you can use this formula to rotate arbitrary vectors in the $\mathbf{i}$-plane over an angle ϕ as

$$\mathbf{R}\mathbf{x} = e^{-\mathbf{i}\phi}\mathbf{x}.$$

Let us try this, using the geometric exponent function `gexp()`:

```
>> i = e1^e2;
>> R = gexp(-i*pi/2);
>> a = e1;
>> draw(a); draw(R*a,'r'); draw(a*R,'g');
```

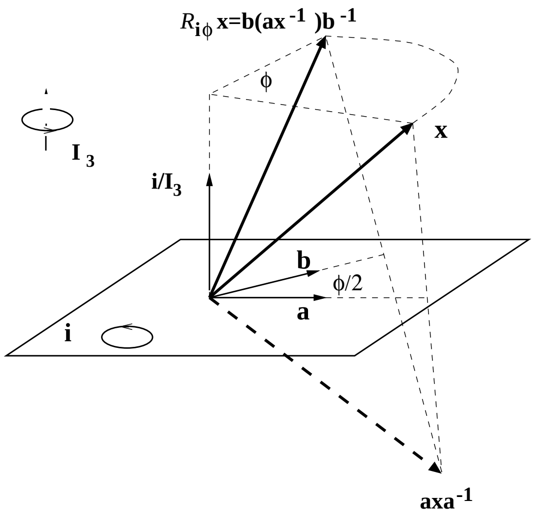

Figure 4: A rotation represented as two reflections.

Note that $\mathbf{Ra}$ indeed rotates $\mathbf{a}$ over $\pi/2$ in the positive orientation direction of the plane $\mathbf{e}_1 \wedge \mathbf{e}_2$ (from $\mathbf{e}_1$ to $\mathbf{e}_2$), and $\mathbf{aR}$ is the opposite orientation (it rotates over $-\pi/2$). If you print `R`, you will notice that there are some numerical issues: it is not the pure bivector it should have been, but this affects the result `R*a` but little. Geometric algebra is numerically stable!

However, if $\mathbf{x}$ is not in the $\mathbf{i}$-plane, the formula $e^{-\mathbf{i}\phi}\mathbf{x}$ does not produce a pure vector. You might try this, for instance with `b=e1+e3`; you will see something a bit surprising. So the above rotation formulas ($\mathbf{Ra}$ and $\mathbf{aR}$) only work for vectors $\mathbf{a}$ in the plane in which $\mathbf{R}$ is defined; i.e., we do not yet have the formula for general rotations in 3-space, which is the topic of the next section.

3.4.2 Rotations as spinors

There is a better representation of those rotations that keep the $\mathbf{i}$-plane invariant, and which will work on *any* vector, whether in the $\mathbf{i}$-plane or not. We construct it by realizing that a rotation can be made by two reflections. We saw in Section 3.3 that the reflection of a vector $\mathbf{x}$ in a vector $\mathbf{a}$ can be written as: $\mathbf{axa}^{-1}$. Following this by a reflection in a vector $\mathbf{b}$ we obtain (take $\mathbf{a}$ and $\mathbf{b}$ as unit vectors):

$$\mathbf{b}(\mathbf{axa}^{-1})\mathbf{b}^{-1} = (\mathbf{ba})\mathbf{x}(\mathbf{ba})^{-1} = e^{-\mathbf{i}\phi/2}\mathbf{x}e^{\mathbf{i}\phi/2},$$

where $\mathbf{i}$ is the plane of $\mathbf{a}$ and $\mathbf{b}$ (so proportional to $\mathbf{a} \wedge \mathbf{b}$) and $\phi/2$ the angle between $\mathbf{a}$ and $\mathbf{b}$. This produces a rotation over ϕ, as you may find by inspecting Figure 4 or writing out the shorthand explicitly. In doing this, it is convenient to split $\mathbf{x}$ in components perpendicular and contained in $\mathbf{i}$, and to use that $\mathbf{x}_\perp\mathbf{i} = \mathbf{i}\mathbf{x}_\perp$ (since $\mathbf{x}_\perp \cdot \mathbf{i} = 0$) and $\mathbf{x}_\parallel\mathbf{i} = -\mathbf{i}\mathbf{x}_\parallel$ (since $\mathbf{x}_\parallel \wedge \mathbf{i} = 0$):

$$\begin{aligned}
e^{-\mathbf{i}\phi/2}\mathbf{x}e^{\mathbf{i}\phi/2} &= (\cos(\phi/2) - \mathbf{i}\sin(\phi/2))\mathbf{x}(\cos(\phi/2) + \mathbf{i}\sin(\phi/2)) \\
&= (\cos(\phi/2) - \mathbf{i}\sin(\phi/2))(\mathbf{x}_\perp + \mathbf{x}_\parallel)(\cos(\phi/2) + \mathbf{i}\sin(\phi/2)) \\
&= (\cos^2(\phi/2) - \mathbf{i}^2\sin^2(\phi/2))\mathbf{x}_\perp + \mathbf{x}_\parallel(\cos^2(\phi/2) - \sin^2(\phi/2) + 2\sin(\phi/2)\cos(\phi/2)\,\mathbf{i}) \\
&= \mathbf{x}_\perp + \mathbf{x}_\parallel(\cos(\phi) + \sin(\phi)\mathbf{i}) \\
&= \mathbf{x}_\perp + \mathbf{x}_\parallel e^{\mathbf{i}\phi}
\end{aligned}$$

So the perpendicular component is unchanged, and the parallel component rotates over ϕ. Therefore the rotation of $\mathbf{x}$ over ϕ in the $\mathbf{i}$-plane is given by the formula

$$\mathbf{x} \to e^{-\mathbf{i}\phi/2}\mathbf{x}e^{\mathbf{i}\phi/2}.$$

This formula represents the desired rotation. Let us try the last example of the previous section (a 90 degree rotation in the $\mathbf{e}_1 \wedge \mathbf{e}_2$-plane) again, now properly using the new formula:

```
>> i = e1^e2;
>> R = gexp(-i*pi/2/2);              %note the half angle!
>> a = e1;
>> b=e1+e3;
>> Ra = R*a/R;
>> Rb = R*b/R;
>> draw(a,'b'); draw(Ra,'b');
>> draw(b,'r'); draw(a^b,'g'); draw(R*(a^b)/R,'g');
```

The new formula easily extends to arbitrary multivectors, as follows. Suppose we want to rotate **cd**. We can perform this rotation by rotating **c** and **d** independently and multiplying the results. But this is simply

$$\left(e^{-\mathbf{i}\phi/2}\mathbf{c}e^{\mathbf{i}\phi/2}\right)\left(e^{-\mathbf{i}\phi/2}\mathbf{d}e^{\mathbf{i}\phi/2}\right) = \left(e^{-\mathbf{i}\phi/2}\mathbf{cd}e^{\mathbf{i}\phi/2}\right).$$

Linearity of the rotation permits us to rotate sums of geometric products using this formula; and since that is all that inner products and outer products are (see Equations 10 and 11), the formula applies to those as well. Therefore, characterizing a rotation **R** by $R = e^{-\mathbf{i}\phi/2}$, we can use it as

$$\mathbf{X} \to R\mathbf{X}R^{-1}$$

to produce the rotated version of **X**, whatever **X** is.

```
>> %... needs a,b,R
>> draw(a^b,'g'); draw(R*(a^b)/R,'g');
>> draw(b^Rb,'m'); draw(R*(b^Rb)/R,'m');
```

The object $R = e^{-\mathbf{i}\phi/2}$ is called a *spinor* (or sometimes a *rotor*). It is all we need to characterize a rotation in n dimensions. Composition of rotations (independent of what we are going to rotate) is then just a multiplication of their spinors. For rotating **X** first by R_1 and then by R_2 gives:

$$R_2(R_1\mathbf{X}R_1^{-1})R_2^{-1} = (R_2R_1)\mathbf{X}(R_2R_1)^{-1}$$

which is a new rotation using the spinor $R = R_2R_1$. Be careful, though: spinors do *not* commute, so you cannot simply say $e^{-\mathbf{i}_2\phi_2}e^{-\mathbf{i}_1\phi_1} = e^{-(\mathbf{i}_2\phi_2+\mathbf{i}_1\phi_1)}$.

Exercises

1. On paper and on the computer, make an operator that implements a rotation of 90 degrees around the $\mathbf{e}_1$ axis, followed by 90 degrees around the $\mathbf{e}_2$ axis. After you have constructed it, write it in exponential form again (on paper only) – this should give you the effective total rotation, in terms of a rotation plane and an angle. (Use `GAview([135 -45])` to bring the view more or less along the effective rotation axis.)
2. Repeat the same exercise using rotation matrices; especially the final step of retrieving axis and angle is quite a bit more work!
3. Draw, rotate and draw a bivector, and make sure your intuition keeps up with these results!
4. What happens when you rotate a trivector? Can you prove that?

3.4.3 Rotations around an axis

In the previous section, we took a rotation in the plane and extended it to a rotation in 3-space. However, we can also characterize rotations in 3-dimensions by an *axis* and an angle. You may wonder how this 3-dimensional rotation is related to the above rotation.

The answer: it is simply related by duality. We can rewrite the bivector $\mathbf{i}\phi/2$ (the rotation plane and angle) as

$$\mathbf{i}\phi/2 = (\mathbf{i}\phi/2)\mathbf{I}_3^{-1}\mathbf{I}_3 = (\text{dual}(\mathbf{i})\phi/2)\mathbf{I}_3 = (\mathbf{u}\phi/2)\mathbf{I}_3 = \mathbf{I}_3\mathbf{u}\phi/2,$$

with $\mathbf{u} \equiv \text{dual}(\mathbf{i})$ the unit vector dual to **i**. It is the *rotation axis.* Thus a rotation in 3D may be characterized by a vector of length $\phi/2$ along the axis **u**:

$$R = e^{-\mathbf{i}\phi/2} = e^{-\mathbf{I}_3\mathbf{u}\phi/2}$$

A very handy formula, all you need is $\mathbf{u}\phi$ to get a computable representation of the rotation! This is natural, much more so than rotation matrices in which the axis is hidden as the eigenvector of eigenvalue 1, and the angle in the complex eigenvalues or the trace. (Beware: in higher dimensions, this rewriting does not work, and you will have to use the bivectors. For instance in 4-dimensional space the dual of the bivector characterizing the rotation plane is a bivector; rotations in 4D do *not* have an axis).

Obviously, all this is similar to the cross product trick, which is also valid in 3D only. We have seen that $\mathbf{a} \times \mathbf{b} = \text{dual}(\mathbf{a} \wedge \mathbf{b})$, so that $\mathbf{u} = \text{dual}(\mathbf{i})$ can be written as the cross product of two vectors in the **i**-plane. However, the characterization of rotation by a bivector works in arbitrary dimensions, and is therefore a good habit even in 3D.

To move back and forth between the spinor in exponential form, and the argument of the exponential, it is convenient to have an *inverse of the exponential*, i.e., a *logarithm*. This inverse is *not* unique, since if $R = exp(\mathbf{i}\phi/2)$, then also $exp(\mathbf{i}\phi/2 + 2\pi\mathbf{i}) = R$; so the logarithm of a spinor has multiple values (those of you familiar with complex analysis may recognize the phenomenon). In the implementation for the logarithm of a spinor we choose the value of the bivector between $-\pi\mathbf{i}$ and $\pi\mathbf{i}$; this is the function `sLog()`. You see it used in the exercise below.[6]

Exercises

1. Verify that the following code is an implementation of the first exercise of the previous section. Note especially how we retrieve the direction of the effective rotation axis.

```
>> %GAblock(2)
>> % ROTATION EXERCISE
>> clf;
>> R1 = gexp(-I3*e1*pi/2/2);
>> R2 = gexp(-I3*e2*pi/2/2);
>> R =  R2*R1;
>> a = e1+e2;
>> Ra = R*a/R;
>> RRa = R*Ra/R;
>> draw(a); draw(Ra,'m'); draw(RRa,'r'); %% Draw the objects
>> axisR = unit( -GAZ(sLog(R))/I3 );
>> draw(axisR,'g'); % Draw the axis of rotation
```

You may be able to visualize things a little better using the bivector of the rotation plane:

```
>> %... needs axisR
>> draw(dual(axisR),'g');
```

3.5 Orientations in 3-space

When a vector, bivector, or other geometric element is rotated relative to another element, we can use that to encode *orientation*. This representation is itself an element of the geometric algebra. Here is how that works, in a Euclidean space.

3.5.1 Interpolation of orientations

Interpolation between orientations in 3-space is a notorious problem; often required, but not easy to do tidily in many a classical representation. In geometric algebra, the problem is fairly easily solved – about as easily as it is stated. First, we characterize orientations in terms of rotation operators, in a coordinate-free manner. The idea is straightforward: when you consider an object to have a certain orientation, what you mean is that relative to its standard orientation (whatever that may be), it has undergone some rotation R. So its orientation may be characterized by this rotation R. If we have two orientations, then these may be characterized by two rotations R_A and R_B. A smooth interpolation between these two orientations may then be achieved by n intermediate *identical* rotation operators R, to be applied to R_A, so that the subsequent rotations are

$$R_0 = R_A; \quad R_{i+1} = R\,R_i; \quad R_n = R_B.$$

The intermediate orientations of whatever object X we wanted oriented are then $R_i X R_i^{-1}$, for $i = 1, \cdots, n$. With this specification of the manner of rotation (n identical operators) the problem can be solved in a fairly straightforward manner.

[6]If you use `sLog` on a non-unit spinor S, then there is an additive scalar term equal to $\log(|S|)$, in agreement with the usual behavior of the logarithm on products.

The total rotation to be made by the n operators R is the rotation from R_A to R_B, which is $R_B R_A^{-1}$. This should equal R^n, so we have

$$R^n = R_B R_A^{-1} = e^{-\mathbf{I}_3 \mathbf{a} \phi/2},$$

where $\mathbf{a}$ and ϕ follow from R_B and R_A (see below for an example). The solution is then

$$R = \left(e^{-\mathbf{I}_3 \mathbf{a} \phi/2}\right)^{1/n} = e^{-\mathbf{I}_3 \mathbf{a} \phi/(2n)}.$$

Taking the n-th root of a spinor can be done by taking its logarithm using `sLog` (giving the argument of the exponential), dividing by n, and then exponentiating.

The following example shows the computations to implement this. Let R_A be the orientation achieved from some standard orientation by a rotation over $\mathbf{I}\mathbf{e}_1\pi/2$, and let R_B be the rotation $\mathbf{I}\mathbf{e}_2\pi/2$. Let us say that the aim is to get from R_A to R_B in eight intermediate steps. We compute:

```
>> %GAblock(3)
>> % INTERPOLATION OF ORIENTATIONS
>> clf;
>> RA = gexp(-I3*e1*pi/2/2);
>> RB = gexp(-I3*e2*pi/2/2);
>> Rtot =  RB/RA
>> n = 8;                          % we rotate in 8 steps
>> R = gexp(sLog(Rtot)/n);
```

To demonstrate the correctness, we need an object that shows the orientations. So we choose to draw some bivector `u^v` in orientation `RA` and `RB`, and show the intermediate orientations.

```
>> %... needs n, R, RA, RB
>> u = e1+e2-e3;
>> v = e1+e3;
>> view = [-0.6  2.5  -1    1.16  -2  1.1];  % select the view
>>       % === initial orientation:
>> DrawBivector(RA*u/RA,RA*v/RA,'b');  axis(view); GAview([30 30]); %%
>>       % === final orientation:
>> DrawBivector(RB*u/RB,RB*v/RB,'g');  axis(view);                 %%
>> axisR = unit(GAZ(-sLog(R)/I3));   % reorientation axis:
>> draw(axisR,'r');                 %% displayed for visualization
>>       % === display of the 7 intermediate orientations
>> Ri = RA;
>> for i=1:n-1
>>     Ri = R*Ri;
>>     ui = Ri*u/Ri;
>>     vi = Ri*v/Ri;
>>     DrawBivector(ui,vi);
>>     drawnow;
>> end
>> GAorbiter(125);
```

This gives a result similar to Figure 5. Make sure you understand the reason behind these steps (the `drawnow` command forces Matlab to display the draw commands in `DrawBivector`; without this command, nothing would be displayed until the `for` loop was exited). If you have learned about *quaternions* before, you may realize that the above geometric algebra approach to interpolating orientation is equivalent to unit quaternions (see also next section). For most people, those are an isolated technique; you may now appreciate how they are a natural part of the algebraic structure of the 3-dimensional space.

Exercises

1. Verify the axis and angle in the example above by hand.
2. What happens in the example if one of the rotations, say R_B, was over 180 degrees, so $\psi = \pi$. Is the solution still well-behaved?
3. What happens if R_A and R_B occur in the same plane?

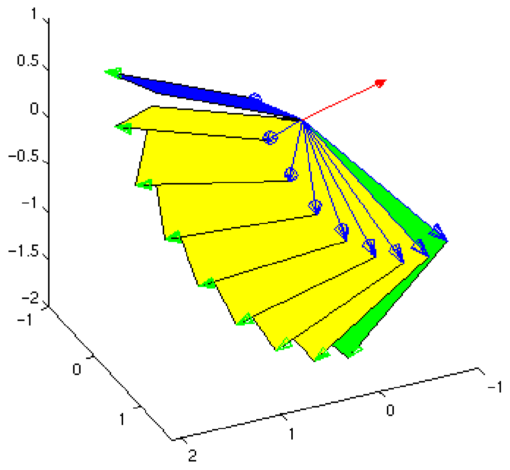

Figure 5: Interpolation of orientations for a bivector.

3.6 Complex numbers and quaternions: subsumed

So rotations are characterized by the plane in which they occur; those planes are characterized by bivectors; and combinations of rotations are represented by geometric products of these bivectors. If you have heard of complex numbers and quaternions, you know that those may also be used to do rotations, in the plane and in space, respectively. How does this compare to the geometric algebra method? Surprisingly, the answer is: it is exactly the same (but in a richer context)!

Algebraically, we may observe that the linear space $\mathcal{C}\ell_{3,0}$ has several interesting subspaces with corresponding sub-algebras. In particular, consider the subspace spanned by $\{1, \mathbf{e}_1 \wedge \mathbf{e}_2\}$ on an orthonormal basis $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$. The product of $\mathbf{e}_1 \wedge \mathbf{e}_2$ with itself is

$$(\mathbf{e}_1 \wedge \mathbf{e}_2)(\mathbf{e}_1 \wedge \mathbf{e}_2) = \mathbf{e}_1\mathbf{e}_2\mathbf{e}_1\mathbf{e}_2 = -\mathbf{e}_1\mathbf{e}_1\mathbf{e}_2\mathbf{e}_2 = -1.$$

Thus in this subspace, the bivector $\mathbf{e}_1 \wedge \mathbf{e}_2$ has similar properties to the complex number i, and we see that indeed we can use this linear subspace of our geometric algebra spanned by 1 and $\mathbf{e}_1 \wedge \mathbf{e}_2$ as the complex numbers, with their product represented by the geometric product (at least if we are careful about the order, due to difference in the commutative properties of the two algebras). Within geometric algebra, these can be seen as operators, which can act on geometrical subspaces represented by vectors, bivectors, trivectors or scalars. Complex numbers can only multiply each other, not 'operate' on other objects such as vectors.

Now consider the subalgebra spanned by $\{1, \mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_2 \wedge \mathbf{e}_3, \mathbf{e}_1 \wedge \mathbf{e}_3\}$. The analysis of the previous paragraph shows that

$$(\mathbf{e}_1 \wedge \mathbf{e}_2)^2 = (\mathbf{e}_2 \wedge \mathbf{e}_3)^2 = (\mathbf{e}_1 \wedge \mathbf{e}_3)^2 = -1,$$

Further, we have

$$\begin{aligned}
(\mathbf{e}_1 \wedge \mathbf{e}_2)(\mathbf{e}_2 \wedge \mathbf{e}_3) &= \mathbf{e}_1 \wedge \mathbf{e}_3 \\
(\mathbf{e}_2 \wedge \mathbf{e}_3)(\mathbf{e}_1 \wedge \mathbf{e}_3) &= \mathbf{e}_1 \wedge \mathbf{e}_2 \\
(\mathbf{e}_1 \wedge \mathbf{e}_3)(\mathbf{e}_1 \wedge \mathbf{e}_2) &= \mathbf{e}_2 \wedge \mathbf{e}_3
\end{aligned}$$

and changing the order in which we multiply these bivectors changes the sign of the result. In other words, with the identification $i = \mathbf{e}_1 \wedge \mathbf{e}_2$, $j = \mathbf{e}_2 \wedge \mathbf{e}_3$, and $k = \mathbf{e}_1 \wedge \mathbf{e}_3$ we get:

$$i^2 = j^2 = k^2 = -1, \quad i\,j = k, \; j\,k = i, \; k\,i = j.$$

You may recognize this: they are the defining equations of *quaternions*, usually introduced as a special 'number system', somehow mysteriously handy for computations with rotations.

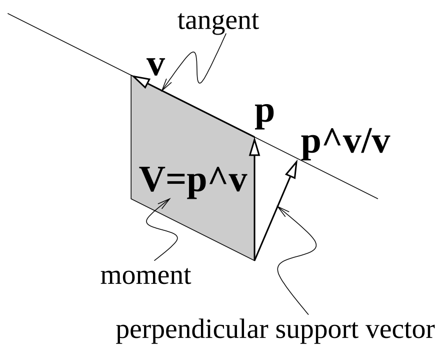

Figure 6: A line described by offset $\mathbf{p}$ and tangent $\mathbf{v}$, with moment $\mathbf{V}$.

Now we see that the basic 'complex numbers' occurring in quaternions are actually a basis of bivectors in the geometric algebra $\mathcal{C}\ell_{3,0}$. Those seemingly imaginary numbers are actually real spatial planes in disguise, and their product is just the geometric product. Not a big surprise, then, that quaternions are so useful in describing orientations: quaternions are actually disguised spinors representing the rotation planes, and their products therefore represent successive rotations. Within the same framework, we can now also represent the objects on which the rotations need to act, and that is a significant improvement over the quaternion representation.

3.7 Subspaces off the origin

3.7.1 Lines off the origin

You may have noticed that most of the objects we have drawn are based about the origin. Vectors start from the origin, bivectors and trivectors contain the origin. It is possible, however, to describe objects that are not centered at the origin, basically by adding a vector, bivector, etcetera to offset them to where we desire. Let's see how.

We have already seen that a blade $\mathbf{B}$ characterizes a subspace through the outer product, as follows: the equation

$$\mathbf{x} \wedge \mathbf{B} = 0$$

holds if and only if $\mathbf{x}$ is linearly dependent on the vectors 'in' the blade. So in particular, if the blade is a vector $\mathbf{v}$, the equation $\mathbf{x} \wedge \mathbf{v} = 0$ characterizes the vectors of points on the line in direction $\mathbf{v}$, through the origin. Its parametric solution is $\mathbf{x} = \tau\mathbf{v}$, for any real number τ.

To characterize a line off the origin, through a point $\mathbf{p}$, we simply translate $\mathbf{x}$, i.e., we let $\mathbf{x} - \mathbf{p}$ satisfy the line equation. This gives

$$\mathbf{x} \wedge \mathbf{v} = \mathbf{p} \wedge \mathbf{v}$$

as the implicit equation of the line parallel to $\mathbf{v}$, through $\mathbf{p}$. So all points $\mathbf{x}$ on the line span the same bivector with $\mathbf{v}$. A line is thus implicitly characterized by a vector $\mathbf{v}$ (called the *tangent*) and a bivector $\mathbf{V} \equiv \mathbf{p} \wedge \mathbf{v}$ (called the *moment*), and determined by an arbitrary vector $\mathbf{p}$ on the line. These terms are illustrated in Figure 6.

This implicit characterization of the line leads to a parametric specification of the points on it, which is

$$\mathbf{x} = (\mathbf{p} \wedge \mathbf{v})/\mathbf{v} + \tau\mathbf{v}.$$

You may recognize in the first term the rejection of $\mathbf{p}$ by $\mathbf{v}$, i.e., the *perpendicular support vector of the line*. It can also be written in terms of the moment as $\mathbf{V}/\mathbf{v}$. The variable τ is a real number; as it varies from $-\infty$ to $+\infty$ we obtain all points on the line.

(For those interested, here is how the parametric equation is obtained from the implicit equation $\mathbf{x} \wedge \mathbf{v} = \mathbf{V}$. First write it as: $\mathbf{x}\mathbf{v} = \mathbf{V} + \mathbf{x} \cdot \mathbf{v}$, then divide out $\mathbf{v}$ to obtain: $\mathbf{x} = \mathbf{V}/\mathbf{v} + (\mathbf{x} \cdot \mathbf{v})/\mathbf{v} = \mathbf{V}/\mathbf{v} + (\mathbf{x} \cdot \mathbf{v}^{-1})\mathbf{v}$. Now recognize that $\mathbf{x} \cdot \mathbf{v}^{-1}$ is a real number taking values between $-\infty$ and $+\infty$, effectively specifying for each point $\mathbf{x}$ its characterizing parameter; call it τ, and you have the parametric equation.)

So for example, if we want a line parallel to $\mathbf{v} = \mathbf{e}_2$ that passes through $\mathbf{p} = \mathbf{e}_1 + \mathbf{e}_2$, we are interested in all vectors $\mathbf{x}$ satisfying the equation $\mathbf{x} \wedge \mathbf{e}_2 = (\mathbf{e}_1 + \mathbf{e}_2) \wedge \mathbf{e}_2 = \mathbf{e}_1 \wedge \mathbf{e}_2$. Obviously

$\mathbf{x} = \mathbf{e}_1 + \tau\mathbf{e}_2$ is the parametric solution, and this is readily verified:

$$\mathbf{x} \wedge \mathbf{e}_2 = (\mathbf{e}_1 + \tau\mathbf{e}_2) \wedge \mathbf{e}_2 = \mathbf{e}_1 \wedge \mathbf{e}_2 = (\mathbf{e}_1 + \mathbf{e}_2) \wedge \mathbf{e}_2,$$

for all real numbers τ. For example, $\mathbf{e}_1 + 2\mathbf{e}_2$ lies on our line. We can test this in GABLE by checking that the equation is satisfied:

```
>> (e1+2*e2)^e2 == (e1+e2)^e2
ans =
     1
```

We can give a graphic illustration of the bivector equality characterizing a line by using `DrawBivector`. First draw the defining bivector for the line, i.e., its moment:

```
>> DrawBivector(e1,e2);
```

Next let's draw the corresponding bivectors for two more pieces of the line:

```
>> %...
>> DrawBivector(e1+e2,e2);
>> DrawBivector(e1+2*e2,e2);
```

What we see in this figure is that the green arrows (which connect points on the line) are collinear.

By varying τ in the parametric equation, we can sample points along the line. We have written a routine called `DrawPolyline()` that will draw the piecewise line segment, in a 'connect the dots' fashion. As arguments, `DrawPolyline` takes a cell array of geometric objects that are all vectors, and draws the piecewise line segment that connects the tips of these vector arguments (in Matlab, a cell array is a list of items enclosed in '{}'). We can use `DrawPolyline` to draw a portion of our line. We begin by clearing the screen and redrawing the bivector representing our line:

```
>> clf; DrawBivector(e1+e2,e2);
```

Next, we use `DrawPolyline` to draw part of the line $\mathbf{x} = \mathbf{e}_1 + \tau\mathbf{e}_2$:

```
>> %...
>> DrawPolyline({e1,e1+3*e2,e1+7*e2});
```

The tips of the three vectors listed should be collinear, as indicated pictorially by `DrawPolyline`. To see that the tips of the vectors `e1+3*e2` and `e1+7*e2` are on the line, draw them using `draw`.

3.7.2 Planes off the origin

For planes, we just use a *bivector* as characterizing the tangent blade $\mathbf{B}$. That gives

$$\mathbf{x} \wedge \mathbf{B} = \mathbf{p} \wedge \mathbf{B}$$

as the equation characterizing all vectors $\mathbf{x}$ of points on the plane with tangent $\mathbf{B}$ and moment $\mathbf{p} \wedge \mathbf{B}$ (i.e., passing through $\mathbf{p}$). We can show this graphically:

```
>> p = e1; b1 = e2; b2 = e3;
>> DrawTrivector(b1,b2,p);
>> DrawTrivector(b1,b2,p+b1);
>> DrawTrivector(b1,b2,p+b2);
>> DrawTrivector(b1,b2,p+b1+b2);
```

This shows pictorially that all these trivectors have a common bivector (which is `b1^b2`), and all the same moment; so the position vectors `p`, `p+b1`, `p+b2`, `p+b1+b2` all lie in the same plane. Use `GAorbiter` and try `GAview([0 60])` to help see that the tips of these four red vectors are coplanar.

You may derive that the perpendicular support vector is now $(\mathbf{p} \wedge \mathbf{B})/\mathbf{B}$.

The parametric equation of the plane involves two scalar parameters (this is a slightly more advanced subject; read the rest of this section only if you are interested). Let $\mathbf{B} = \mathbf{b}_1 \wedge \mathbf{b}_2$ with $\mathbf{b}_1$ and $\mathbf{b}_2$ orthogonal so that we may even write $\mathbf{B} = \mathbf{b}_1\mathbf{b}_2$, then starting from the

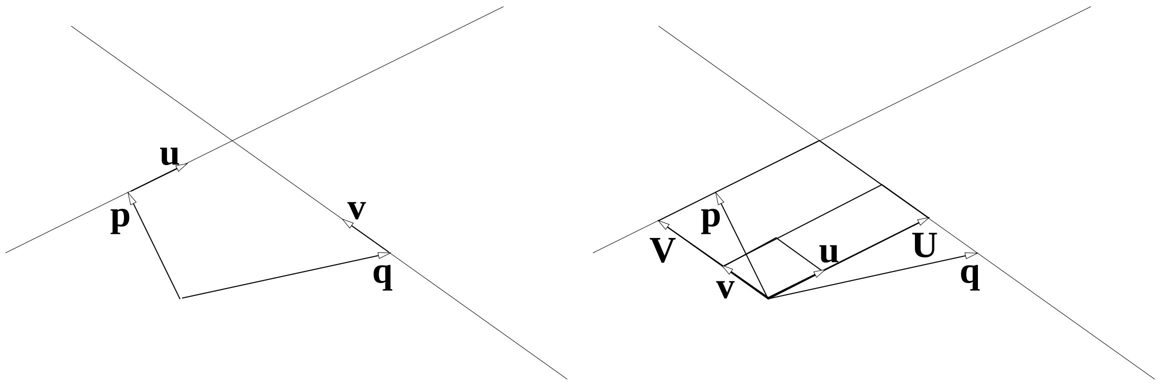

Figure 7: Intersection of two lines.

parametric equation derived similarly to that of the line, we rewrite using the inner product identity $\mathbf{x}\cdot(\mathbf{b}_1\mathbf{b}_2) = (\mathbf{x}\cdot\mathbf{b}_1)\mathbf{b}_2 - (\mathbf{x}\cdot\mathbf{b}_2)\mathbf{b}_1$ (which is equation 13):

$$\begin{aligned}
\mathbf{x} &= (\mathbf{x}\wedge\mathbf{B})/\mathbf{B} + (\mathbf{x}\cdot\mathbf{B})/\mathbf{B} \\
&= (\mathbf{p}\wedge\mathbf{B})/\mathbf{B} + (\mathbf{x}\cdot\mathbf{B})/\mathbf{B} \\
&= (\mathbf{p}\wedge\mathbf{B})/\mathbf{B} + (\mathbf{x}\cdot(\mathbf{b}_1\mathbf{b}_2))\mathbf{B}^{-1} \\
&= (\mathbf{p}\wedge\mathbf{B})/\mathbf{B} + ((\mathbf{x}\cdot\mathbf{b}_1)\mathbf{b}_2 - (\mathbf{x}\cdot\mathbf{b}_2)\mathbf{b}_1)\,\mathbf{B}^{-1} \\
&= (\mathbf{p}\wedge\mathbf{B})/\mathbf{B} + (\mathbf{x}\cdot\mathbf{b}_1)\mathbf{b}_2\mathbf{B}^{-1} - (\mathbf{x}\cdot\mathbf{b}_2)\mathbf{b}_1\mathbf{B}^{-1} \\
&= (\mathbf{p}\wedge\mathbf{B})/\mathbf{B} + (\mathbf{x}\cdot\mathbf{b}_1)\mathbf{b}_1^{-1} + (\mathbf{x}\cdot\mathbf{b}_2)\mathbf{b}_2^{-1} \\
&= (\mathbf{p}\wedge\mathbf{B})/\mathbf{B} + (\mathbf{x}\cdot\mathbf{b}_1^{-1})\mathbf{b}_1 + (\mathbf{x}\cdot\mathbf{b}_2^{-1})\mathbf{b}_2 \\
&= (\mathbf{p}\wedge\mathbf{B})/\mathbf{B} + \tau_1\mathbf{b}_1 + \tau_2\mathbf{b}_2,
\end{aligned}$$

with the appropriate definitions of $\tau_1 \equiv (\mathbf{x}\cdot\mathbf{b}_1^{-1})$ and $\tau_2 \equiv (bx\cdot\mathbf{b}_2^{-1})$.

3.7.3 Intersection of two lines

Suppose we have two lines in a plane (but not through the origin) and we wish to find their intersection. For example, as illustrated in the left half of Figure 7, suppose we have the two lines described by

$$\begin{aligned}
\mathbf{x}\wedge\mathbf{u} &= \mathbf{p}\wedge\mathbf{u} \\
\mathbf{x}\wedge\mathbf{v} &= \mathbf{q}\wedge\mathbf{v}
\end{aligned}$$

In this figure, we draw **p** and **q** emanating from the origin, and **u** and **v** emanating from the heads of **p** and **q** respectively to indicate the direction of the lines.

Rearranging to get the diagram on the right half of this figure, we see that the intersection is given by $\mathbf{U}+\mathbf{V}$, where **U** and **V** are scaled versions of **u** and **v** respectively. What we need to find are the two scale factors. But from the figure, we see that the scale factor needed to scale **u** to **U** is given by the ratio of the area of the parallelogram defined by **U** and **v** (which is $\mathbf{q}\wedge\mathbf{v}$) to the area of the parallelogram defined by **u** and **v** (which is $\mathbf{u}\wedge\mathbf{v}$). These quantities thus come immediately from the outer product, and we using the symmetric construction to get the scale factor for **V**, we see that the intersection is given by

$$\frac{\mathbf{q}\wedge\mathbf{v}}{\mathbf{u}\wedge\mathbf{v}}\mathbf{u} + \frac{\mathbf{p}\wedge\mathbf{u}}{\mathbf{v}\wedge\mathbf{u}}\mathbf{v}. \tag{17}$$

Note here that the outer products in each ratio are bivectors whose area is the area of the parallelogram spanned by the two vectors. When we divide a bivector by a parallel bivector, the bivector parts cancel, leaving only the ratio of the two scalar components. Here is an example of application of Equation 17:

```
>> %GAblock(4)
>> % LINE INTERSECTS LINE
>> p = e2; u = 0.2*e2 + e1;
>> q = e1; v = e2-2*e1;
>> clf;
>> draw(p); GAview([0 90]);
>> DrawPolyline({p-2*u,p+2*u});
>> draw(q,'g'); DrawPolyline({q-v,q+2*v},'g');
```

In this diagram, we have drawn two lines, one in blue and one in green. Now draw

```
>> %...
>> U = (q^v/(u^v)) * u
>> V = (p^u/(v^u)) * v
>> draw(U,'m')          %% Draw U
>> draw(V, 'm')         %% Draw V
>> draw(U+V, 'r' )      % Draw U+V
```

Here we see the magenta vectors are scaled versions of the `u` and `v` directions of the lines. The sum of the magenta vectors is the red vector, which points to the intersection point of the two lines.

Exercises

1. Intersect the line $\mathbf{x} \wedge (\mathbf{e}_1 + 2\mathbf{e}_2) = 0$ with the line $\mathbf{x} \wedge (\mathbf{e}_2 - \mathbf{e}_1) = (\mathbf{e}_1 + \mathbf{e}_2) \wedge (\mathbf{e}_2 - \mathbf{e}_1)$. Draw a picture similar to the one above.
2. What happens to the above intersection code if one of the lines passes through the origin (e.g., `p=0;`)? Does it still work? Why or why not?
3. What happens to the above intersection code if the lines are parallel (i.e., $\mathbf{u} = \mathbf{v}$)? Try this in GABLE. Later we will see line intersection code that does not have this problem.
4. As a guess for the formula specifying the intersection point of a line $\mathbf{x} \wedge \mathbf{u} = \mathbf{p} \wedge \mathbf{u}$ and a plane $\mathbf{x} \wedge \mathbf{V} = \mathbf{q} \wedge \mathbf{V}$ (where $\mathbf{V}$ is the bivector tangent of the plane), try:
$$\frac{\mathbf{q} \wedge \mathbf{V}}{\mathbf{u} \wedge \mathbf{V}}\mathbf{u} + \frac{\mathbf{p} \wedge \mathbf{u}}{\mathbf{V} \wedge \mathbf{u}}\mathbf{V}$$
on some examples. Did it work? Can you see why, geometrically?

3.8 Summary

In this section, we have studied geometric computations. Using the three products, we have seen how to compute projections, rejections, reflections and rotations. We also looked at lines off the plane, and saw how to use the geometric operations to intersect these lines.

4 Blades and subspace relationships

Blades are the fundamental objects in geometric algebra. We have seen how a k-blade is formed by taking the outer product of k independent vectors. We have also shown that a blade represents a k-dimensional subspace containing the origin. In the next section, we present a trick to make blades represent subspaces off-set from the origin. That obviously makes blades very relevant objects in geometric algebra, and their geometry worthy of a separate study.

In this section, we determine the operations required to assess the relative positions of blades: how they project onto each other, how they can be decomposed relative to each other, and how they intersect. This intersection will necessitate the definition of a new kind of product in geometric algebra, the `meet`, and a related product, the `join`, as geometric union.

4.1 Subspaces as blades

Picture, in your mind, two blades; they can be a vector and a bivector, two vectors, two bivectors, or they may even be scalars (blades of grade 0). As you visualize them in the way we have done, you actually make each of those k-blades characterize a k-dimensional subspace, through the equivalence

vector $\mathbf{x}$ *in subspace determined by* $\mathbf{A}$ $\quad\Leftrightarrow\quad \mathbf{x} \wedge \mathbf{A} = 0.$

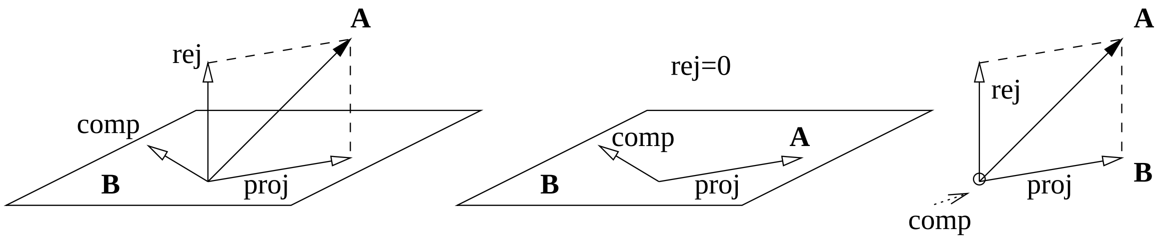

Figure 8: Subspace **A** and its geometric relationship relative to a subspace **B**, for several relative grades and positions.

This interpretation of blades is not specific on sign or magnitude of **A**, so by themselves, subspaces have no significant sign or magnitude. For now, we can think of subspaces as 'blades modulo a multiplicative scalar' – but we'll keep in mind that geometric algebra would allow us to be more quantitative about subspaces than we may have been used to. By the way, note that a scalar blade **A** yields the equation $\mathbf{x} = 0$, so that it represents a point at the origin.

We would like to encode the following relationships of subspaces into geometric algebra:

- *projection* of subspaces onto each other, and its counterpart: *rejection* of subspaces by each other
- *intersection of subspaces*, including how 'orthogonal' this intersection is
- *'union' of subspaces*: common span, or common superspace, with some measure of their degree of parallelness
- *the 'sides' of a space split by a subspace*
- *a distance measure between subspaces*

We will see that a limited set of operators can express all these concepts, and that they are composed of the products we have defined so far.

4.2 Projection, rejection, orthogonal complement

Consider the examples of the subspaces sketched in Figure 8. It depicts subspaces **A** and **B** of various grades, and some subspaces derived from their geometric relationships. We will now find the corresponding formulas.

We have already mentioned before, in Section 3.1, that the projection of a blade **A** onto a blade **B** is given by the formula

$$\mathrm{proj}(\mathbf{A},\mathbf{B}) \equiv (\mathbf{A}\cdot\mathbf{B})/\mathbf{B}.$$

We interpret this as the projection formula for subspaces: it gives the component of **A** entirely in **B**.[7] This projection is the inner product divided by **B**, which is like the dual in the **B**-plane of the inner product. The inner product $\mathbf{A}\cdot\mathbf{B}$ by itself therefore also has geometric significance: it is the orthogonal complement, within **B**, of the projection of **A**. We will denote it by $\mathrm{comp}(\mathbf{A},\mathbf{B})$ if we mean to imply this meaning. You may want to think of it as 'the part of **B** not contributing to **A**',

The 'perpendicular' part of **A** to **B** is the *rejection*, it is the component of **A** *not* in **B**:

$$\mathrm{rej}(\mathbf{A},\mathbf{B}) = \mathbf{A} - \mathrm{proj}(\mathbf{A},\mathbf{B}) = (\mathbf{AB} - \mathbf{A}\cdot\mathbf{B})/\mathbf{B} \qquad (18)$$

Only when **A** is a vector **a** can we write this as $\mathrm{rej}(\mathbf{a},\mathbf{B}) = \mathbf{a} - \mathrm{proj}(\mathbf{a},\mathbf{B}) = (\mathbf{aB} - \mathbf{a}\cdot\mathbf{B})/\mathbf{B} = (\mathbf{a}\wedge\mathbf{B})/\mathbf{B}$, and even then there are problems in 1-dimensional space; it is better to use the universal Equation 18.

So the relationship of two subspaces **A** and **B** split into three parts:

- The projection $\mathrm{proj}(\mathbf{A},\mathbf{B})$ of **A** onto **B**, which equals $(\mathbf{A}\cdot\mathbf{B})/\mathbf{B}$.
- The rejection $\mathrm{rej}(\mathbf{A},\mathbf{B})$ of **A** by **B**, which is $\mathbf{A} - \mathrm{proj}(\mathbf{A},\mathbf{B})$; it projects to zero, which we may express by saying it is the part of **A** *perpendicular* to **B**.

[7]If you use the Hestenes inner product, you have to perform an extra test to make sure that you have the blade with smallest grade as first argument, or this interpretation will not work. Our use of the contraction as inner product requires no such testing.

projection proj$(\mathbf{A},\mathbf{B})$ $= (\mathbf{A}\cdot\mathbf{B})/\mathbf{B}$	rejection rej$(\mathbf{A},\mathbf{A})$ $= \mathbf{A} - (\mathbf{A}\cdot\mathbf{B})/\mathbf{B}$	'complement' comp$(\mathbf{A},\mathbf{B})$ $\mathbf{A}\cdot\mathbf{B}$	description
scalar	0	ℓ-blade	**A** is a scalar
k-blade	0	$(\ell-k)$-blade	**A** lies in **B**
k-blade	0	scalar	**A** and **B** coincide
k-blade	k-blade	$(\ell-k)$-blade	$\ell \neq k$: in general position
k-blade	k-blade	scalar	non-trivial intersection
0	k-blade	0	**A** perpendicular to **B**

Table 2: Relationship between two subspaces **A** (of grade k) and **B** (of grade $\ell \geq k$).

- The complement comp$(\mathbf{A},\mathbf{B})$, which is the part of **B** orthogonal to **A** and its projection; this is the subspace $\mathbf{A}\cdot\mathbf{B}$. This is a new concept; you see how it is an atom in the construction of the projection. (Proof of orthogonality to **A** is easy using Equation 14: $\mathbf{A}\cdot(\mathbf{A}\cdot\mathbf{B}) = (\mathbf{A}\wedge\mathbf{A})\cdot\mathbf{B} = 0\cdot\mathbf{B} = 0$.)

Depending on how they relate, some of these subspaces may be zero, or identical to **A**. Such events define the various relative situations of **A** and **B**, see Table 2 and Figure 8.

Let us play with this in 3 dimensions:

```
>> A = e1-e2/2+e3;      % a line
>> B = e1^(e2+e3/2);    % a plane
>> cAB = inner(A,B);
>> pAB = cAB/B;
>> rAB = A - pAB;
>> clf; draw(A,'b'); draw(B,'g');
>> draw(cAB,'r'); draw(pAB,'c'); draw(rAB,'m');
```

(use `GAorbiter` for interpretation!). Now replace `B` by the line `e1/2+e2+e3/3` (or a line of your choice) and run the same commands. Note that projection and rejection are as expected, and that the orthogonal complement to `A` in `B` is now a scalar (printed in red above the figure). The point at the origin is indeed the only subspace of `B` and 'orthogonal' to both `A` and its projection.

Now do the same on two non-identical planes, for instance `A = e2^(e1+e3)` and `B = e1^(e2+e3/2)`, and interpret the results. (As you may see, the magnitudes of the bivectors are somewhat difficult to interpret using the `draw` command. We'll soon use `DrawBivector` instead, but then we first need to find the intersection of the planes, to span them nicely.)

This subspace interpretation casts an interesting new light on the inner product of vectors. We have seen that a scalar is, geometrically, the representation of a point at the origin. The inner product $\mathbf{u}\cdot\mathbf{v}$ of two vectors **u** and **v** is the subspace of **v** which is the orthogonal complement of **u**; and geometrically, that is the point at the origin, on the line determined by **v**. So the inner product of two vectors is a weighted point...

If you want to stretch your intuition for generalization to new limits, use the space itself for `B`, so take `B = I3` and take `A` to be a line, and then a plane. Verify what you see by computation! Why does the dual subspace appear? Can you make yourself see and feel that the rejection of `A` from 3-space is 0, which is 'not even a point'?

The same formulas and interpretation are still applicable when `A` and `B` are coincident spaces: play around with `B = 2*A`, choosing a line, plane or `I3` for `A`. What is the projection now?

What you may learn from all this is that *geometric algebra has no special cases* for computing such geometric relationships. It is not always easy to find the characterization in words of what is computed since it may not be a classically recognized concept (our notion of 'orthogonal complement of **A** in **B**' is a bit of a stretch), but we have found it rewarding to add the outcomes of straightforward computations as a elementary concepts to our intuition.

4.3 Angles and distances

A vector **x** has several relevant relationships with respect to a planar or linear subspace **A** in 3-space such as its incidence angle, the distance of its tip to the subspace, and the 'side' it lies on. With projection and rejection, these are easy to compute.

- *perpendicular distance*
  The perpendicular distance is obviously the length of the rejection $|(\text{rej}(\mathbf{x}, \mathbf{A}))| = |(\mathbf{x} - (\mathbf{x} \cdot \mathbf{A})/\mathbf{A})| = |((\mathbf{x} \wedge \mathbf{A})/\mathbf{A})|$.
- *side*
  The quantity $\mathbf{x} \wedge \mathbf{A}$ is the space spanned by $\mathbf{x}$ and $\mathbf{A}$; we can compare this to the blade we might already have for this subspace to determine whether $\mathbf{x}$ lies on the 'positive' or 'negative' side of $\mathbf{A}$. If $\mathbf{A}$ is a 2-blade, then $\mathbf{x} \wedge \mathbf{A}$ is a 3-blade (or zero, in which case $\mathbf{x}$ lies *in* $\mathbf{A}$). So we can compute the *side* as the sign of the scalar $(\mathbf{x} \wedge \mathbf{A})/\mathbf{I}_3$. If $\mathbf{A}$ is one-dimensional, there is no natural orientation to give to the 2-blade $\mathbf{x} \wedge \mathbf{A}$: so a line in 3-space has no sides (for a point – it does induce a sense in lines).
  Note that the side is closely related to the perpendicular distance: for the plane we can write $\mathbf{I}_3 = \mathbf{n}\mathbf{A}$, with $\mathbf{n}$ the normal vector of $\mathbf{A}$ (so that $\mathbf{n} \cdot \mathbf{A} = 0$). Then $(\mathbf{x} \wedge \mathbf{A})\mathbf{I}_3^{-1} = (\mathbf{x} \wedge \mathbf{A})/\mathbf{A}/\mathbf{n} = \text{rej}(\mathbf{x}, \mathbf{A})/\mathbf{n}$. Since the rejection and $\mathbf{n}$ are parallel, this is a scalar, expressing the perpendicular distance in units of $\mathbf{n}$ (if you take $\mathbf{x} = \mathbf{n}$, you get 1). So for a point relative to a plane, we obtain a signed distance, and that is often more useful than just the absolute value of the perpendicular distance.
- *angle*
  The angle of a vector relative to a line or plane is characterized by the ratio between its rejection and projection, which is like the tangent function from trigonometry:
  $$\frac{\text{rej}(\mathbf{x}, \mathbf{A})}{\text{proj}(\mathbf{x}, \mathbf{A})} = \frac{\mathbf{x} \wedge \mathbf{A}}{\mathbf{x} \cdot \mathbf{A}}.$$
  The magnitudes of $\mathbf{x}$ and $\mathbf{A}$ cancel, so it is properly independent of scale. Note that we have not taken norms, so that the result is a *bivector* (it is the ratio of a $(k+1)$-blade and a $(k-1)$-blade, so a 2-blade). We may write the outcome as $\mathbf{i} \tan \phi$. This explicitly gives us the plane $\mathbf{i}$ in which the angle is to be measured, and it cancels the bothersome ambiguity of sign which the use of a tangent always involves. The above expression computes $\mathbf{i} \tan \phi$, and if you wished to interpret the angle in the oppositely oriented plane $-\mathbf{i}$, the value of the tangent function (and therefore the angle ϕ) automatically changes sign. As we have seen when treating rotations, angles are essentially bivectors! (It is even possible to extend the trigonometric functions naturally to geometric algebra; then the formula may be said to compute $\tan(\mathbf{i}\phi)$, see [7]).

Let us try this out:

```
>> % GAblock(5)
>> % PROJECTION, REJECTION
>> x = e1 + e2/2+e3;
>> A = e2 + e3/3;   % a linear subspace
>> xA = geoall(x,A);
>> clf; draw(x,'b'); draw(A,'g');       %% Draw A and x
>> DrawPolyline({xA.rej,xA.rej+xA.proj,xA.proj},'k');
>> draw(xA.rej,'m'); draw(xA.proj,'m'); %% Draw rej and proj
>> distxA = norm(xA.rej)
>> tanglexA = xA.rej/xA.proj
>> anglexA = atan(norm(tanglexA))*180/pi %%
>> B = e1^A;             % A planar subspace (containing A)
>> xB = geoall(x,B);
>> draw(B,'y');          %%
>> DrawPolyline({xB.rej,xB.rej+xB.proj,xB.proj},'k');
>> draw(xB.rej,'r'); draw(xB.proj,'r'); %% rej and proj for B
>> distxB = norm(xB.rej)
>> tanglexB = xB.rej/xB.proj
>> anglexB = atan(norm(tanglexB))*180/pi
>> side = sign((x^B)/I3)
```

By the way, the GABLE code `GA(0)` creates the geometric zero object.

Exercises

1. What do you get if you compute these quantities relative to a 0-dimensional subspace $\mathbf{A} = 2$, or relative to the total 3-space $\mathbf{A} = \mathbf{I}_3$? Can you interpret those outcomes sensibly?

2. If you add '1' to the bivector we computed above, you get an operator which (under post-multiplication) rotates and scales $\mathbf{x}$ to be in the subspace $\mathbf{A}$. Why? Check this for the subspaces in the GABLE example above.
3. In 3-dimensional space, the relative orientation of two planes is also determined by an angle. Can you give a formula for the value of its tangent function?

4.4 Intersection and union of subspaces: `meet` and `join`

In the example of the two planes:

```
>> % GAblock(6)
>> % MEET, JOIN
>> A = e2^(e1+e3);
>> B = e1^(e2+e3/2);
>> cAB = inner(A,B);  pAB = cAB/B; rAB = A - pAB;
>> clf; draw(A,'b'); draw(B,'g'); %%
>> draw(cAB,'r'); draw(pAB,'c'); draw(rAB,'m');
```

we see that the formulas so far do not determine their line of intersection. Such an intersection between subspaces is obviously something we often need; so we need to make it computational.

First, how do we determine that a subspace $\mathbf{A}$ has a non-trivial intersection with subspace $\mathbf{B}$? In such a case, there is at least a 1-dimensional subspace $\mathbf{c}$ in common. Rewriting the blades in the form $\mathbf{A} = \mathbf{A}' \wedge \mathbf{c}$ and $\mathbf{B} = \mathbf{c} \wedge \mathbf{B}'$, we see that $\mathbf{A} \wedge \mathbf{B} = (\mathbf{A}' \wedge \mathbf{c}) \wedge (\mathbf{c} \wedge \mathbf{B}') = \mathbf{A}' \wedge (\mathbf{c} \wedge \mathbf{c}) \wedge \mathbf{B}' = \mathbf{A}' \wedge 0 \wedge \mathbf{B}' = 0$. So we find:

$$\mathbf{A} \textit{ and } \mathbf{B} \textit{ have non-trivial intersection} \quad \Leftrightarrow \quad \mathbf{A} \wedge \mathbf{B} = 0.$$

Of course the fact that $\mathbf{A} \wedge \mathbf{B} = 0$ may mean that there are several independent 1-dimensional subspaces in common; $\mathbf{A}$ and $\mathbf{B}$ might even coincide completely. Let us suppose that $\mathbf{C}$ is a *largest common subspace* of $\mathbf{A}$ and $\mathbf{B}$ (in the sense of: the highest dimensionality, so the highest grade). Then we can write $\mathbf{A} = \mathbf{A}' \wedge \mathbf{C}$ and $\mathbf{B} = \mathbf{C} \wedge \mathbf{B}'$, so we can 'factor out' the common subspace $\mathbf{C}$. (The weird order is to get better signs later on.)

The common subspace $\mathbf{C}$ is the 'intersection' of the two subspaces $\mathbf{A}$ and $\mathbf{B}$. We would like to define an operation in geometric algebra that produces it; let us call that the `meet` operation. It is thus defined by:

If $\mathbf{A}$ *and* $\mathbf{B}$ *can be factored using a highest grade sub-blade* $\mathbf{C}$ *as* $\mathbf{A} = \mathbf{A}' \wedge \mathbf{C}$ and $\mathbf{B} = \mathbf{C} \wedge \mathbf{B}'$*, then their* `meet` *is* $\mathtt{meet}(\mathbf{A}, \mathbf{B}) = \mathbf{C}$.

We have indeed implemented such an operation:

```
>> %... needs A,B
>> mAB = meet(A,B)
>> clf; draw(A,'b'); draw(B,'g'); draw(mAB,'y');
```

However, you should be somewhat careful with using its outcome. The problem is that the factorization is not unique, and so neither is the `meet`. Since $\mathbf{C}$ is a blade indicating a subspace, any multiple $\gamma\mathbf{C}$ is just as valid, and that would produce a `meet` of $\gamma\mathbf{C}$. We could demand $|\mathbf{C}| = 1$ (at least in Euclidean spaces), but that would still leave the sign undetermined. The outcome of the `meet` is therefore a strange 'object': it has no well-defined sign. Its meaning as a *subspace* is perfectly clear ($\mathbf{x}$ is in $\mathbf{C}$ iff $\mathbf{x} \wedge \mathbf{C} = 0$), but as a *blade* it is ambiguous: it has no unique orientation that can be established on the basis of $\mathbf{A}$ and $\mathbf{B}$ alone.

If we consider $\mathbf{C}$ in the context of the superspace spanned by $\mathbf{A}$ and $\mathbf{B}$, we can be more specific on its orientation. This common superspace is like a *union* of the spaces $\mathbf{A}$ and $\mathbf{B}$. It is often called the `join` of $\mathbf{A}$ and $\mathbf{B}$, and we define it through the factorization we had before:

If $\mathbf{A}$ *and* $\mathbf{B}$ *can be factored using a highest grade sub-blade* $\mathbf{C}$ *as* $\mathbf{A} = \mathbf{A}' \wedge \mathbf{C}$ and $\mathbf{B} = \mathbf{C} \wedge \mathbf{B}$*, then their* `join` *is:* $\mathtt{join}(\mathbf{A}, \mathbf{B}) = \mathbf{A}' \wedge \mathbf{C} \wedge \mathbf{B}' \quad (= \mathbf{A}' \wedge \mathbf{B} = \mathbf{A} \wedge \mathbf{B}')$.

This we have also implemented:

```
>> %... needs A,B
>> jAB = join(A,B)
>> draw(jAB,'k');
```

Obviously, in this case of two distinct 2-blades in 3-space, it is proportional to $\mathbf{I}_3$. If $\mathbf{A}$ and $\mathbf{B}$ would be disjoint, then $\texttt{join}(\mathbf{A},\mathbf{B})$ is proportional to $\mathbf{A} \wedge \mathbf{B}$.

Again, the problem is that the factorization is not unique, and replacing $\mathbf{C}$ by $\gamma\mathbf{C}$ now changes the `join` by a factor of $1/\gamma$. Thus we have the problem of assigning a magnitude to the `join`, as we did for the `meet`. Although we have no mechanism in geometric algebra to fix the magnitudes uniquely (and this is a fundamental impossibility, since it is the ambiguity of the factorization that causes it), we can at least make a `meet` and `join` that are consistent, in the sense that they are based on the same factorization. This can be done based on the observation that `join` and `meet` are related by

$$\texttt{join}(\mathbf{A},\mathbf{B}) = \frac{\mathbf{A}}{\texttt{meet}(\mathbf{A},\mathbf{B})} \wedge \mathbf{B}.$$

(A derivation may be found in [12, 13].) It can be rewritten to the more commonly seen form

$$\texttt{meet}(\mathbf{A},\mathbf{B}) = \frac{\mathbf{B}}{\texttt{join}(\mathbf{A},\mathbf{B})} \cdot \mathbf{A}.$$

We have moreover scaled the `meet` to have unit norm. This fixes `meet` and `join` up to one shared ambiguous sign.

4.5 Combining subspaces: examples

With projection, rejection, orthogonal complement, `meet` and `join`, we have a powerful set of terms to describe many (all?) relationships between subspaces. And they are all computationally related, we can use their properties both synthetically to construct new objects reflecting the relationships, or analytically in derivations.

Let us play around with these new tools. It is convenient to have a function that computes all the quantities, so that you can execute it whenever you have defined new objects (using cut-and-paste, or the scroll in Matlab, or as a `.m`-file), and a function to draw them all in standard colors. So we use what we had before, but make it into a function that computes the things in a single structure:

```
function r = geoall(A,B)
   r.obj1 = A;
   r.obj2 = B;
   r.comp = inner(A,B);
   r.proj = r.comp/B;
   r.rej  = A - r.proj;
   r.meet = meet(A,B);
   r.join = join(A,B);

function drawall(r)
   clf; draw(r.obj1,'b'); draw(r.obj2,'g');
   draw(r.comp,'r'); draw(r.proj,'c'); draw(r.rej,'m');
   draw(r.meet,'k'); draw(r.join,'y');
```

These functions have been precoded for your convenience (in the actual code, we have included some argument protection – testing for blades – to make them more robust). Note the colors used to draw objects, and realize that if objects overlap exactly, only the last one drawn is visible. It is therefore a good idea to use non-unit-norm blades for `A` and `B`, so that the various objects are more likely to have different magnitudes.

Let us first continue the example of the two intersecting planes $\mathbf{A}$ and $\mathbf{B}$ of Section 4.2. Remember how it was hard to see whether the projection was done correctly when we drew the bivectors using `draw`? We can use the above functions to redo that quickly:

```
>> A = e2^(e1+e3);
>> B = e1^(e2+e3/2);
>> clf; drawall(geoall(A,B));
```

Now that we have the `meet` operation, we can use the `meet` as a common vector in the two planes, and this permits us to use `DrawBivector` to show the various blades more clearly in a factored form.

```
>> % GAblock(7)
>> % MEET AND JOIN DECOMPOSED
>> A = e2^(e1+e3);
>> B = e1^(e2+e3/2);
>> aAB = geoall(A,B);
>> M = aAB.meet;
>> clf;
>> DrawBivector(A/M,M,'b');          % draw A, decomposed
>> DrawBivector(M,1/M*B,'g');        %% draw B, decomposed
>> DrawBivector(aAB.proj/M,M,'c');   % draw proj, decomposed
>> DrawBivector(aAB.rej/M,M,'m');    %% draw rej, decomposed
>> draw(aAB.comp,'r');
>> draw(aAB.meet,'k');
>> draw(aAB.join,'y');
```

(use `GAorbiter(30,2)` repeatedly to interpret the result – note how this partially shows up the projection in cyan). You now see how the magnitudes of projection and rejection obviously give the original `A` as their sum, which was hard to appreciate in the circular way of drawing the bivectors.

What we have defined is general for blades of any dimension. For instance, let's take a 1-blade and a 2-blade.

```
>> figure       % pop up a new figure for comparison with previous
>> A = e1-e2/2+e3;
>> B = e1^(e2+e3/2);
>> drawall(geoall(A,B));
```

The black caption `scalar = 1` shows that the intersection is now the point at the origin.

Or take a 0-blade and a 1-blade, a rather degenerate case:

```
>> figure
>> A = 3;
>> B = 2*(e2+e3);
>> drawall(geoall(A,B));
```

(By the way, if you want to go back to drawing in the window for previous figure `k`, type `figure(k)`.)

No matter how degenerate the situation, our relationship operators have no exceptions, and the results are always sensibly interpretable. These functions and operators are extremely stable to any blades you might use them on, and they can be used to implement all of classical geometry. One more step is needed: to show how blades can represent off-set subspaces such as lines and planes in general position. We do that in the next section.

Exercises

1. The vector $\mathbf{p} = \mathbf{e}_1 + \mathbf{e}_2$ lies in the plane $\mathbf{A} = (\mathbf{e}_1 + \mathbf{e}_2) \wedge \mathbf{e}_3$ (as you may verify). How far does its endpoint lie from the intersection of $\mathbf{A}$ with the plane $\mathbf{B} = 2(\mathbf{e}_1 \wedge (\mathbf{e}_2 + \mathbf{e}_3)$? Draw this!
2. What is the angle between $\mathbf{p}$ and the intersection of $\mathbf{A}$ and $\mathbf{B}$?

5 The homogeneous model

We have focused on blades, which represent subspaces through the origin. In many applications, you are of course interested in subspaces translated off the origin, such as tangent blades. In Section 3.7 we treated those by implicit equations, and that is rather indirect. We will show now that blades can represent such 'off-set subspaces' – but we have to use blades from a geometric algebra of one dimension higher. We first create the intuition by using the pinhole model of imaging, and then dive into the mathematics.

5.1 The geometry of imaging: the pinhole camera

Consider the geometry of imaging, which is a geometrical problem common for many application fields such as computer graphics, computer vision and robotics. To simplify the physical issues

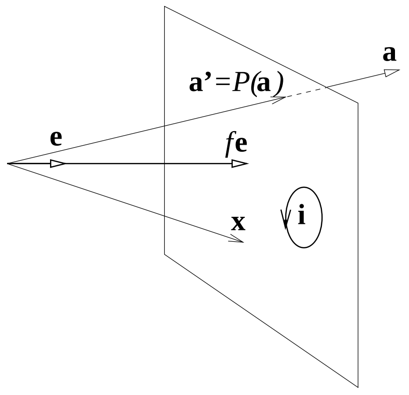

Figure 9: Pinhole camera model.

of imaging, we use a simple *pinhole model* for our camera, in which light rays are straight lines passing through a point hole, and imaged by intersection with an *image plane*. Although this image plane is behind the hole at the distance of the focal length, we may equivalently draw the image in front of the hole at the same distance, as is customary in those fields of application.

So consider Figure 9; the focal point is at $f\mathbf{e}$ (with $\mathbf{e}$ a unit vector), the carrier of the image plane is perpendicular to $\mathbf{e}$ and is therefore a multiple of dual($\mathbf{e}$); we denote it by $\mathbf{i}$. This image plane is the set of points $\mathbf{x}$ whose inner product with $\mathbf{e}$ equals the focal length f. We will take $f = 1$, so the image plane points satisfy:

$$\mathbf{x} \cdot \mathbf{e} = 1.$$

A point $\mathbf{a}$ in the world is imaged onto the image plane in the direction $\mathbf{a}$: $P(\mathbf{a}) \equiv \mathbf{a}'$. Thus $\mathbf{a}'$ satisfies the two equations $\mathbf{a}' \wedge \mathbf{a} = 0$ and $\mathbf{a}' \cdot \mathbf{e} = 1$; solving this system of equations using Equation 12 gives

$$0 = \mathbf{e} \cdot (\mathbf{a}' \wedge \mathbf{a}) = \mathbf{a} - (\mathbf{e} \cdot \mathbf{a})\mathbf{a}', \quad \text{so that} \;\; \mathbf{a}' = \frac{\mathbf{a}}{\mathbf{a} \cdot \mathbf{e}}.$$

This is what you would expect: the vector is rescaled to have inner product 1 (the focal length) with the focal point vector $\mathbf{e}$, to make it lie in the image plane.

Here is how we draw this in GABLE:

```
>> %GAblock(8)
>> % PINHOLE IMAGING
>> clf;
>> e = e1;
>> %==== indicate focal point and image plane:
>> draw(e,'k');
>> IP = {e1+e3+e2,e1+e3-e2,e1-e3-e2,e1-e3+e2,e1+e3+e2}; % image plane rectangle
>> DrawPolygon(IP,'y')          %% The image plane
>> %==== vector x and its projection
>> x = 3*e1 - 1.5*e2 + 2*e3;
>> px = x/inner(x,e);
>> draw(x,'b');                 %% Draw x
>> DrawSimplex({px},'r','r');
```

(The `DrawSimplex` routine with one vector argument between the parentheses draws that vector in the first specified color, with a thickly drawn point at its tip in the second color specified.)

Any vector on the line $\mathbf{x} \wedge \mathbf{a} = 0$ is of course mapped onto the same point $P(\mathbf{a})$, so this is a projection of Euclidean 3-space onto a 2-dimensional space. The *ray* in space maps onto a *point* of the image plane.

In the pinhole projection, a 3-space line projects onto line on a 2-dimension space. While we could project the 3-space line point-by-point, it is more interesting to determine its complete 2-space geometric representation. We derive that now, but let us first draw the figure that results so that you can visualize better what is going on in the derivation.

```
>> %... needs e, IP, x, px
>> y = 1.8*e1 + 1.5*e2 + e3;
>> u = y-x;
>> U = x^y;
>> d = U/u;
>> clf; draw(e1,'k'); DrawPolygon(IP,'w'); % Redraw e and the image plane
>> DrawPolygon({GA(0),x,y},'y');           % Draw Polygon
>> draw(d,'g'); draw(x); draw(y);          %% Draw x and y and support
>> DrawSimplex({x,y},'n','g');             %% Draw line
>> uprime = inner(e1,U);
>> dprime = U/uprime;
>> draw(dprime,'r');                    %% Draw proj support
>> DrawSimplex({px,px+unit(uprime)},'n','r'); %% Projected line
>> DrawSimplex({e1,dprime},'n','m');    %% Perpendicular support in plane
>> GAview([-30 10])
```

Use `GAorbiter(20,2)` repeatedly to get the 3-D feeling of the situation, as you go through the explanation. Remember that the GABLE code `GA(0)` creates the geometric zero object.

Points on a line in space are characterized by the line's tangent $\mathbf{u}$ (in green) and moment $\mathbf{U}$ (the yellow triangle indicates half the moment), as $\mathbf{x} \wedge \mathbf{u} = \mathbf{U}$ (Figure 6), or parametrically as $\mathbf{x} = \mathbf{U}/\mathbf{u} + \tau\mathbf{u}$. The vector $\mathbf{U}/\mathbf{u}$ is the perpendicular support vector (also in green). This 1-dimensional subspace of 3-space gets projected to a line in the image plane. The projected tangent vector of the line into the image plane we denote by $\mathbf{u}'$ (in red). It needs to satisfy $\mathbf{u}' \cdot \mathbf{e} = 0$ and $\mathbf{u}' \wedge \mathbf{U} = 0$: $\mathbf{u}'$ is in $\mathbf{U}$, perpendicular to $\mathbf{e}$. It follows immediately from this geometric formulation that

$$\mathbf{u}' = \mathbf{e} \cdot \mathbf{U}$$

or any positive multiple of it. The moment of this line in the image plane (viewed as a 3-dimensional line) is obviously still proportional to $\mathbf{U}$ (after all, this line is in the yellow triangle as well), so the spatial line in the plane has the implicit equation

$$\mathbf{x} \wedge \mathbf{u}' = \mathbf{U}.$$

A position vector on it is the perpendicular to it, computed as usual by

$$\mathbf{d}' = \frac{\mathbf{U}}{\mathbf{u}'} = \frac{\mathbf{U}}{\mathbf{e} \cdot \mathbf{U}}.$$

This is a vector in 3-space (in red in the figure). It can be written as the sum of $\mathbf{e}$ and a vector in the tangent to the image plane $\mathbf{i}$: $\mathbf{d}' = \mathbf{e} + \mathbf{d}$. This $\mathbf{d}$ (in magenta) is the actual perpendicular support vector of the projected line in the image plane. The implicit equation of the projected line in the image plane is therefore

$$\mathbf{y} \wedge \mathbf{u}' = \mathbf{d} \wedge \mathbf{u}',$$

(we use $\mathbf{y}$ for the image plane vector) or, after some substituting $\mathbf{u}'$ and $\mathbf{d}$,

$$\mathbf{y} \wedge (\mathbf{e} \cdot \mathbf{U}) = \mathbf{U} - \mathbf{e} \wedge (\mathbf{e} \cdot \mathbf{U}).$$

So its tangent, moment and support are all determined fully by $\mathbf{U}$ only. The tangent $\mathbf{u}$ of the original line in 3-space is *not* important: it gets 'projected out'. This is all obvious from the figure: once you know the yellow triangle (which denotes $\mathbf{U}$) and the image plane (determined by $\mathbf{e}$), the projection of the line is fully determined.

In the figure we have drawn on the screen, you see that the tangent of the projected line corresponds properly to the tangent of the original; it has the same sense of direction. (Note that we rescaled `uprime`: its magnitude is related to $\mathbf{U}$ and can be rather large. It is *not* possible to relate it to the magnitude of the projection of $\mathbf{u}$ in a constant manner, since that projected magnitude varies along the projected line.)

So we find, with those expressions for $\mathbf{u}'$ and $\mathbf{d}'$, that the pinhole projection of a line in 3 dimensions onto a plane depends only on the moment 2-blade $\mathbf{U}$ of that 3-space line. Reversing this construction, now for a general plane (not just the image plane of a pinhole camera), we get that

each line ('off-set 1-space') in the plane is determined by a 2-blade $\mathbf{U}$ *in a 3-dimensional space.*

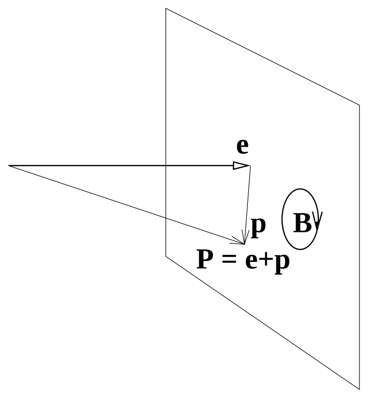

Figure 10: Homogeneous representations.

Of course any multiple of **U** works just as well. This trick works for lines, and also for other off-set subspaces, of any dimension. For instance, we have also seen how a ray through the origin, characterized by a 3-dimensional vector **u**, becomes a point in the plane at the position characterized by the planar vector $\mathbf{u}/(\mathbf{e} \cdot \mathbf{u}) - \mathbf{e}$. Therefore *each point ('off-set 0-space') in the plane is determined by a vector (i.e., a 1-blade) in 3-dimensional space.*

The pinhole model thus gives us the intuition that lines, whether in space or in the image plane, can be represented by 2-blades, and similarly for other subspaces. (The coefficients of such a 2-blade in 3-space are actually the Plücker coordinates of the line, giving another natural embedding of an extraneous trick.) In the geometric algebra treatment of *projective geometry*, this representation of lines in space is exploited – but we will not get into that in this tutorial (you might check [6] for more on this subject). Instead, we focus completely on the image plane, and use the blade representation of lines there to give a treatment of some of the elements of *affine geometry* in geometric algebra, namely the subspaces off the origin. This is known as the *homogeneous model* of 2-dimensional Euclidean space.

5.2 The homogeneous model in $\mathcal{C}\ell_{3,0}$ of 2-space

Obviously, off-set subspaces are useful in many geometric settings. They are in fact what we usually describe as points, lines, etcetera; we always mean that those objects are translatable subspaces. However, common usage identifying terms from vector algebra with these objects tends to be a bit sloppy. Often people equate 'point' and 'vector', whereas actually a vector is a measured *direction*, and *not* a point (which is an object, to which we could apply that vector as a translation). Yet we saw in the previous section that points and vectors are closely related, so it is no surprise that they 'feel similar': points can be represented as vectors 'in 1 more dimension'. The way we ran into that was fairly *ad hoc*, through studying imaging. Let us now formalize these representational issues independent of such a physical motivation.

The basic procedure is this: to represent n-dimensional space with off-set subspaces, we embed it in an $(n + 1)$-dimensional space. We do so by extending the space with an 'extra vector' **e** perpendicular to the n-space. Then we get the following representation of our k-dimensional subspaces off the origin:

- *point*
  A point **P** in n-space, at location **p**, is represented by the $(n + 1)$-space vector:
  $$\mathbf{P} = \mathbf{e} + \mathbf{p}.$$
  In the pinhole model, this is the vector characterizing the ray projecting to the point **P**.
- *direction vector*
  The difference between two such points is a direction vector in the plane. Therefore direction vectors in n-space are represented by $(n+1)$-space vectors with zero **e**-component. Such vectors are perpendicular to **e**, so a direction vector is represented by
  $$\mathbf{v} \text{ such that } \mathbf{e} \cdot \mathbf{v} = 0$$

- *direction bivector*
  A 2-dimensional direction of the n-space (i.e., the tangent of a plane in n-space) is represented as a 2-blade in $(n+1)$-space, without an $\mathbf{e}$ component.
  $$\mathbf{B} \text{ such that } \mathbf{e}\cdot\mathbf{B} = 0$$
- *higher grade tangents*
  Obviously, a direction blade (tangent) of grade k is represented by a k-blade in $(n+1)$-space without $\mathbf{e}$-component.
- *line*
  A line in n-space, with direction $\mathbf{v}$ and some given point $\mathbf{P}$ on it is represented as the $(n+1)$-space 2-blade
  $$\mathbf{P}\wedge\mathbf{v} = (\mathbf{e}+\mathbf{p})\wedge\mathbf{v} = \mathbf{e}\wedge\mathbf{v}+\mathbf{p}\wedge\mathbf{v}.$$
  Note that this contains the tangent $\mathbf{v}$ and the moment $\mathbf{p}\wedge\mathbf{v}$, in a manner in which we can retrieve them (simply by taking the inner product with $\mathbf{e}$). This is indeed a line, for if we ask which points $\mathbf{x}$ are on it, we can resolve this by checking whether the *representation* of the points $\mathbf{x}$ (which is $\mathbf{e}+\mathbf{x}$) are contained in the bivector:
  $$0 = (\mathbf{e}+\mathbf{x})\wedge(\mathbf{e}\wedge\mathbf{v}+\mathbf{p}\wedge\mathbf{v}) = \mathbf{e}\wedge\mathbf{p}\wedge\mathbf{v}+\mathbf{x}\wedge\mathbf{e}\wedge\mathbf{v} = \mathbf{e}\wedge(\mathbf{p}\wedge\mathbf{v}-\mathbf{x}\wedge\mathbf{v})$$
  and since $\mathbf{e}$ is independent of the n-space vectors, this retrieves the implicit line equation
  $$\mathbf{x}\wedge\mathbf{v} = \mathbf{p}\wedge\mathbf{v},$$
  which we used to describe the line in Section 3.7. The representation of a line by a 2-blade is more elegant, and simplifies line intersection. (As we will soon see, you just use the `meet` to intersect them.)
- *higher dimensional subspaces*
  In n-space the higher dimensional directions and affine subspaces are constructed in a similar way: the tangent blade $\mathbf{V}$ is represented as $\mathbf{V}$, a point $\mathbf{P}$ at $\mathbf{p}$ by $\mathbf{e}+\mathbf{p}$, and the affine subspace through $\mathbf{p}$ with tangent $\mathbf{V}$ is represented as the blade
  $$\mathbf{A} = \mathbf{P}\wedge\mathbf{V} = (\mathbf{e}+\mathbf{p})\wedge\mathbf{V} = \mathbf{e}\wedge\mathbf{V}+\mathbf{p}\wedge\mathbf{V}.$$
  (Note that a point $\mathbf{P} = \mathbf{e}+\mathbf{p} = (\mathbf{e}+\mathbf{p})\wedge 1$ can be viewed as a 0-dimensional subspace with as tangent the 0-blade 1 (or any multiple); somewhat strange, but consistent.)

Any of these representations is curiously unspecific on the location that generated them: $\mathbf{p}$ cannot be retrieved from $\mathbf{A} = (\mathbf{e}+\mathbf{p})\wedge\mathbf{V}$. If there is a need to give a point on them to determine how they are off-set in space, it is convenient to use the perpendicular support vector $\mathbf{d}$. We saw in Section 3.7.1 that this vector can be computed as the moment divided by the tangent, and both of those *can* be retrieved from the homogeneous representation $\mathbf{A}$ by some clever manipulation (nice exercise!):
$$\mathbf{d} = \frac{\mathbf{p}\wedge\mathbf{V}}{\mathbf{V}} = \frac{\mathbf{e}\cdot(\mathbf{e}\wedge\mathbf{A})}{\mathbf{e}\cdot\mathbf{A}}.$$
We have implemented a drawing routine that uses the homogeneous representation of off-set subspaces. In the same notation as the general case discussed above, we have

```
>> % GAblock(9)
>> % HOMOGENEOUS REPRESENTATION
>> e = e3;                                % direction of extra dimension
>> clf; draw(e,'k')
>> p = e1+e2/2;
>> P = e+p;             % point representation
>> U = 1;
>> DrawHomogeneous(e,P^U,'y','b')   %% point
>> V = e2-e1/3;
>> DrawHomogeneous(e,P^V,'y','r')   %% line segment
>> W = e1^e2;
>> DrawHomogeneous(e,P^W,'y','g')   % plane segment
```

(The two color arguments have the same function as in `DrawSimplex`. In particular, you can use as first color `'n'` (to avoid drawing the carrier) and then use `GAview([0 90])` to see the 2-space by itself.) Note that the line through `p` in the direction `V` is *not* drawn starting at `p`, but at the perpendicular support vector (which is the only objectively retrievable point).

5.3 Creating geometric objects: a line as the `join` of points

Now that we have the homogeneous model in which all subspaces are represented by pure blades, we can think in a more structured manner about the creation of geometric objects: we can make many of them by the union and intersection of blades, i.e., through `meet` and `join`. In particular, let us revisit our representation of lines.

Consider a line through two distinct points P (at location $\mathbf{p}$) and Q (at location $\mathbf{q}$). In the homogeneous model, these points are represented by $\mathbf{P} = \mathbf{e} + \mathbf{p}$ and $\mathbf{Q} = \mathbf{e} + \mathbf{q}$. The line through them is represented by a bivector; could this be $\mathbf{P} \wedge \mathbf{Q}$? Let us verify:

$$\mathbf{P} \wedge \mathbf{Q} = (\mathbf{e} + \mathbf{p}) \wedge (\mathbf{e} + \mathbf{q}) = \mathbf{e} \wedge \mathbf{q} + \mathbf{p} \wedge \mathbf{e} + \mathbf{p} \wedge \mathbf{q} = \mathbf{e} \wedge (\mathbf{q} - \mathbf{p}) + \mathbf{p} \wedge \mathbf{q}$$

Now introduce the direction vector of the line as the difference between two points on it: $\mathbf{u} \equiv \mathbf{q} - \mathbf{p}$. Observe that $\mathbf{p} \wedge \mathbf{q} = \mathbf{p} \wedge (\mathbf{q} - \mathbf{p}) = \mathbf{p} \wedge \mathbf{u} \equiv \mathbf{U}$, with $\mathbf{U}$ the moment of the line in n-space. So the above reads

$$\mathbf{P} \wedge \mathbf{Q} = \mathbf{e} \wedge \mathbf{u} + \mathbf{p} \wedge \mathbf{u} = \mathbf{P} \wedge \mathbf{u}$$

which is precisely the homogeneous representation of the line. Therefore we indeed find that

> *the homogeneous representation of the line through two distinct points is the outer product of their homogeneous representations*

What happens if the points are identical? Obviously, the outer product becomes zero, and we could accept this as: there is no line connecting two identical points. However, in many situations we would like to return the point itself, as the degenerate object now defined by the 'connect-the-dots' intuition.

Knowing that the points are represented by blades, we might try the `join` operation on those as the implementation of this generalized 'connect-the-dots'. And indeed, the `join` of two distinct blades is proportional to their outer product, covering the case above, whereas the join of identical blades is proportional to either of them. So we have the more general, and more robust:

> *the homogeneous representation of the object obtained by connecting two points is the* `join` *of their homogeneous representations*

Let's try this:

```
>> % GAblock(10)
>> % HOMOGENEOUS BIVECTOR REPRESENTATION OF A LINE
>> clf;
>> e = e3;         % the extra dimension of the homogeneous embedding
>> p = e1/3+e2;    % position of point P
>> q = e1+e2/2;    % position of point Q
>> P = e+p;        % homogeneous representation of P
>> Q = e+q;        % homogeneous representation of Q
```

Now that we have defined points `P` and `Q`, we can construct the line between them with the `join` operation:

```
>> %... needs P,Q
>> PQ = join(P,Q);       % homogeneous representation of the line join(P,Q)
>> DrawSimplex({P},'y','b'); DrawSimplex({Q},'y','b');
>> draw(PQ,'g')          %% the 2-blade PQ drawn as a planar tangent
>> DrawBivector(P,Q,'y')                  %% the 2-blade redrawn
>> DrawSimplex({P,Q},'n','r');            %  the line segment PQ
>> DrawSimplex({e,e+2*e1,e+2*e2},'n','w'); %% the plane of 2-space
```

This plot shows both the 2-blade in 3-space determining the line, and a segment of the line in the plane $\mathbf{x} \cdot \mathbf{e} = 1$. But it is not fair to represent the points `P` and `Q` defining the line, for the homogeneous representation is not specific on location; the 2-blade drawn in green shows that clearly. A better representation is to draw an objective position on the line (for instance, its perpendicular support vector $\mathbf{d}$) and its direction $\mathbf{v}$. These can be retrieved from the homogeneous representation (remember how? if not, see Section 5.2). `DrawHomogeneous` does just this.

```
>> % GAblock(11)
>> % HOMOGENEOUS REPRESENTATION OF OFFSET SUBSPACES
>> e = e3;         % the extra dimension of the homogeneous embedding
>> p = e1/3+e2;     % position of point P
>> q = e1+e2/2;     % position of point Q
>> P = e+p;         % homogeneous representation of P
>> Q = e+q;         % homogeneous representation of Q
>> PQ = join(P,Q);        % homogeneous representation of the line join(P,Q)
>> clf;  draw(e,'k');
>> DrawHomogeneous(e,P,'y','b'); GAtext(1.07*P,'P'); % Draw P
>> DrawHomogeneous(e,Q,'y','g'); GAtext(1.07*Q,'Q') %% Draw Q
>> DrawHomogeneous(e,PQ,'y','r');
>> GAorbiter(30,2);
```

which still shows the 3-D embedding, but with less clutter. We can see the line direction in red, 'carried' by the yellow vectors. Of course when we do computations in the plane, the embedding is irrelevant and we prefer a view that does not show it.

```
>> %...
>> IP = {e-e1/2-e2/2,e+3/2*e1-e2/2,e+3/2*e1+3/2*e2,e-e1/2+3/2*e2};
>> DrawPolygon(IP,'w');
>> GAview([0 90]);
```

Now try this for identical points:

```
>> %... needs P
>> DrawHomogeneous(e,join(P,P),'n','b');
```

This shows the point `P` resulting from the connection of `P` to `P`. Or how about this:

```
>> %... needs P,Q
>> r = e1/2-e2/4;
>> R = e+r;
>> DrawHomogeneous(e,join(join(P,Q),R),'n','g');
```

This shows the plane connecting the three points `P`, `Q`, and `R`. Only the area is significant, since any plane in 2-space contains any point of that space. Let us draw all these objects defined by connecting three points in all their significantly different permutations:

```
%...
   >> %... needs P,Q,R
   >> clf;
   >> DrawHomogeneous(e,P,'n','b');
   >> DrawHomogeneous(e,Q,'n','b');
   >> DrawHomogeneous(e,R,'n','b');
   >> DrawHomogeneous(e,join(P,Q),'n','r');
   >> DrawHomogeneous(e,join(Q,R),'n','r');
   >> DrawHomogeneous(e,join(R,P),'n','r');
   >> DrawHomogeneous(e,join(join(P,Q),R),'n','g');
```

Note again that the position is not present in any of the homogeneous objects of grade higher than 1, but that the magnitude correctly represents the connection length (for lines) or area (for the triangular simplex). Thus the representation of line is indeed of a 'measured direction along a particular line' without being specific on position, and the representation of a plane is a 'measured area'.

We have thus found that the `join` operation in geometric algebra, applied in a homogeneous representation, is the precise computational equivalent of joining the geometrical objects involved. This makes for a convenient computational language: the definition `join(P,Q)` of the segment of the segment determined by `P` and `Q` is simultaneously the computational symbol with all the proper properties. Perhaps surprisingly, those do *not* involve the locations `P` and `Q` themselves. A line only acquires definite points through intersection with other geometrical objects, as the next section shows.

5.4 Creating geometric objects: a point as the `meet` of lines

With off-set spaces represented as pure blades, it should be clear that their intersection should be related to the `meet` of these blades. And indeed, continuing the previous example we can retrieve the points as the intersection of two lines, even though each of those lines do not contain that position explicitly anymore.

```
>> % GAblock(12)
>> % LINE INTERSECTION AS MEET OF HOMOGENEOUS BIVECTORS
>> clf;
>> e = e3;             % extra dimension of the homogeneous embedding
>> P = e+ e1/3+e2;     % point P
>> Q = e+ e1+e2/2;     % point Q
>> R = e+ e1/2-e2/4;   % point R
>> PQ = join(P,Q);     % line PQ
>> QR = join(Q,R);     % line QR
>> meet(PQ,QR)         % intersection of those lines
```

So even though neither `PQ` nor `QR` contains the point `Q` explicitly, their combination retrieves it neatly. A drawing shows this:

```
>> %... needs P,Q,R,PQ,QR
>> clf;
>> draw(e,'k');
>> GAview([0 90]);
>> DrawHomogeneous(e,P,'n','b'); GAtext(1.07*P,'P'); %  P
>> DrawHomogeneous(e,Q,'n','b'); GAtext(1.07*Q,'Q'); %  Q
>> DrawHomogeneous(e,R,'n','b'); GAtext(1.07*R,'R'); %% R
>> DrawHomogeneous(e,PQ,'n','c'); %% PQ
>> DrawHomogeneous(e,QR,'n','g'); %% QR
>> DrawHomogeneous(e,meet(PQ,QR),'n','m'); % Meet of PQ, QR
```

So indeed, *the* `meet` *acts as the intersection operator* in the homogeneous representation.

It is insightful to show this in the 3-dimensional space of the representing blades:

```
>> %... needs P,Q,R,PQ,QR
>> GAview([30 15]);
>> DrawHomogeneous(e,P,'y','b'); %  P
>> DrawHomogeneous(e,Q,'y','b'); %  Q
>> DrawHomogeneous(e,R,'y','b'); % R
>> DrawHomogeneous(e,meet(PQ,QR),'n','m'); %% Meet of PQ, QR
>> draw(PQ,'c'); draw(QR,'g'); draw(meet(PQ,QR),'m'); %%
>> IP = {e-e1-e2,e-e1+2*e2,e+2*e1+2*e2,e+2*e1-e2};
>> DrawPolygon(IP,'w');
```

(use `GAorbiter` to interpret!). This shows the two 2-blades representing the lines (in cyan and green) and their `meet` (in magenta). The interpretation in the 2-space of the `meet` is precisely the point `Q`.

5.5 Distances between offset subspaces

(Warning: this section goes a bit far, and can be skipped.)

Using the homogeneous representation, we can also retrieve the quantitative aspects of relationships between objects. In general in 3-dimensional space, you need a translation and a rotation to bring objects to coincide as much as they will. For instance, two lines can be translated to have a common point, and then rotated/scaled to have their tangents coincide.

In the homogeneous model, translating an object $\mathbf{A} = (\mathbf{e} + \mathbf{p}) \wedge \mathbf{V}$ over a vector $\mathbf{t}$ to a new location $\mathbf{p} + \mathbf{t}$ to make $\mathbf{A}' = (\mathbf{e} + \mathbf{p} + \mathbf{t}) \wedge \mathbf{V}$ is done simply by adding the moment $\mathbf{t} \wedge \mathbf{V}$ to $\mathbf{A}$:

translation of $\mathbf{A}$ *over* $\mathbf{t}$*: add* $\mathbf{t} \wedge (\mathbf{e} \cdot \mathbf{A})$.

Let us call this the *translational moment* of $\mathbf{t}$ for $\mathbf{A}$. This has all the right properties: translating a line over a vector $\mathbf{t}$ is obviously only sensitive to the component of $\mathbf{t}$ perpendicular to the line.[8] Verify that the above has this property!

We can retrieve the translation required to bring $\mathbf{A}$ into the origin of 2-space (i.e., minus its moment) from $\mathbf{A}$. It is

$$-\mathrm{rej}(\mathbf{A}, \mathbf{e} \wedge (\mathbf{e} \cdot \mathbf{A})) = -\mathbf{e} \cdot (\mathbf{e} \wedge \mathbf{A}).$$

[8]For those of you in image analysis, this is the solution to the 'aperture problem' in optic flow, which has difficulty in retrieving a translation vector for a moving straight edge, partially seen. The problem was in the representation, not in the observation.

The left hand side should be understood as follows: we span a blade $\mathbf{e} \wedge (\mathbf{e} \cdot \mathbf{A})$, using the vector $\mathbf{e}$ (pointing to the origin) and the tangent of $\mathbf{A}$ (which is $\mathbf{e} \cdot \mathbf{A}$); then the rejection of $\mathbf{A}$ by that blade is fully in 2-space, and is the moment of $\mathbf{A}$. You may want to verify that the right hand side is indeed $\mathbf{p} \wedge \mathbf{V}$.

Let us compute the translation required to bring $\mathbf{A}$ into a non-trivial intersection with a second offset subspace $\mathbf{B}$, possibly of a different grade than $\mathbf{A}$. We first need to determine the common elements in the tangent directions $\mathbf{e} \cdot \mathbf{A}$ and $\mathbf{e} \cdot \mathbf{B}$ of $\mathbf{A}$ and $\mathbf{B}$, since we want to measure the translation perpendicular to such directions (for example, two parallel lines have a distance measured perpendicular to their direction). So we determine the `join` of the directions, and divide out their `meet`; together this gives the non-common parts of their tangents. Then, as before, we construct a blade through $\mathbf{e}$ containing this tangent:

$$\mathbf{U} \equiv \mathbf{e} \wedge \frac{\texttt{join}(\mathbf{e} \cdot \mathbf{A}, \mathbf{e} \cdot \mathbf{B})}{\texttt{meet}(\mathbf{e} \cdot \mathbf{A}, \mathbf{e} \cdot \mathbf{B})}.$$

The translational moment required to bring $\mathbf{A}$ to coincide with this in the origin is $-\mathrm{rej}(\mathbf{A}, \mathbf{U})$; the perpendicular translation vector corresponding to that is $-\mathrm{rej}(\mathbf{A}, \mathbf{U})/(\mathbf{e} \cdot \mathbf{A})$. To then get to $\mathbf{B}$ we need the translation vector $\mathrm{rej}(\mathbf{B}, \mathbf{U})/(\mathbf{e} \cdot \mathbf{B})$. Combining these to obtain the desired translational moment for $\mathbf{A}$, we get

The translational moment for $\mathbf{A}$ *to make the* `meet` *with* $\mathbf{B}$ *non-trivial is:*

$$\left(\frac{\mathrm{rej}(\mathbf{B}, \mathbf{U})}{\mathbf{e} \cdot \mathbf{B}} - \frac{\mathrm{rej}(\mathbf{A}, \mathbf{U})}{\mathbf{e} \cdot \mathbf{A}} \right) \wedge (\mathbf{e} \cdot \mathbf{A}),$$

where $\mathbf{U} = \mathbf{e} \wedge \texttt{join}(\mathbf{e} \cdot \mathbf{A}, \mathbf{e} \cdot \mathbf{B})/\texttt{meet}(\mathbf{e} \cdot \mathbf{A}, \mathbf{e} \cdot \mathbf{B})$.

This is universally valid, independent of the grade of $\mathbf{A}$ and $\mathbf{B}$. You may check that the unknown scalar factors in `meet` and `join` cancel in this formula (since the resulting blade is used in a rejection, which is insensitive to scalar factors), so that it yields a proper object in geometric algebra, without ambiguity in scale or orientation.

We have implemented this formula as `connection(e,A,B)`, for the translational moment from `A` to `B`. Note that the interchange of `A` and `B` will in general give an object of different grade. Let's play with this:

```
>> %GAblock(13)
>> % CONNECTIONS OF LINES AND POINTS
>> clf;
>> e = e3;
>> P = e+ e1/4+e2/2;
>> Q = e- e1/2+e2/4;
>> R = e+ e1/3-e2/5;
>> PQ = P^Q;
>> PtoQ  = connection(e,P,Q);
>> RtoPQ = connection(e,R,PQ)   % a vector!
>> PQtoR = connection(e,PQ,R)   %% a bivector!
>> draw(e,'k');
>> DrawHomogeneous(e,P,'y','b'); GAtext(1.07*P,'P'); % P
>> DrawHomogeneous(e,Q,'y','b'); GAtext(1.07*Q,'Q'); % Q
>> DrawHomogeneous(e,R,'y','b'); GAtext(1.07*R,'R'); %% R
>> DrawHomogeneous(e,PQ,'n','g');           %% PQ
>> DrawSimplex({P,P+PtoQ},'y','r');         %% segment P to Q
>> DrawSimplex({R,R+RtoPQ},'y','m');        %% segment R to PQ
>> DrawHomogeneous(e,R+RtoPQ,'n','k');      %% closest point on PQ
>> DrawHomogeneous(e,PQ+PQtoR,'n','k');     %% closest line through R
>> IP = {e-e1-e2,e-e1+e2,e+e1+e2,e+e1-e2};
>> DrawPolygon(IP,'w');
```

It also works for parallel lines:

```
>> %... needs P,R,PtoQ
>> Pline = P^PtoQ;    % the line PQ: at P, tangent PtoQ
>> Rline = R^PtoQ;    % the line at R parallel to PQ
>> DrawHomogeneous(e,Rline,'n','c');         %% Rline drawn
>> linecon = connection(e,Pline,Rline);
>> DrawHomogeneous(e,Pline+linecon,'n','m'); %% translated Pline
```

For intersecting lines, the connection is zero, since nothing needs to be added to make their `meet` non-trivial:

```
>> %... needs PQ,Q,R
>> QR = join(Q,R);
>> intersect = meet(PQ,QR)/inner(e,meet(PQ,QR));
>> linecon = connection(e,PQ,QR);
>> DrawSimplex({intersect,intersect+linecon},'y','k');
```

(Note the black result: a scalar.) Again, the geometric algebra formula has no exceptions, it works for all objects whatever their dimensionality. This means that there is no use for any conditional statements in our programs – this simpler code is much less error-prone!

To determine the angle between the objects **A** and **B** at their point of intersection, you can use the techniques of Section 4.3 directly on the tangent blades ($\mathbf{e} \cdot \mathbf{A}$) and ($\mathbf{e} \cdot \mathbf{B}$).

5.6 Summary: the grammar of Euclidean geometry

With the homogeneous model, we have the power to embed much of Euclidean geometry directly into geometric algebra. This is the way it works in general in n-dimensional spaces:

- Any off-set subspace is represented by a pure blade in $(n + 1)$-space. Notably, points (off-set 0-spaces) are represented by vectors in $(n + 1)$-space.
- Creating a k-dimensional off-set subspace by connecting a point to a $(k - 1)$-dimensional subspace, is represented by the `join` of a 1-blade to a k-blade (a line is the `join` of two points, etc.). And in general, you can connect ℓ-dimensional spaces to k-dimensional spaces using the `connection` operation on their blade representations.
- Intersection of the off-set subspaces is represented by the `meet` of their representative blades (the intersection point of a line is their `meet`, etc.).

It is therefore now possible to give a computationally prescriptive meaning to such symbols as '**PQ**' for the line determined by the points **P** and **Q**: represent the points by the appropriate homogeneous 1-blades, and read the symbol '**PQ**' as `join`(**P**, **Q**), the 2-blade representation of the line. (If you know the points are disjoint, you may use the outer product: so then $\mathbf{P} \wedge \mathbf{Q}$ represents the line through the points **P** and **Q**.) The permissible algebraic manipulations of these symbols then follow from the laws of geometric algebra. People actually use this for automatic theorem proving, see [1]. In this sense, geometric algebra provides the *grammar of geometry*, and it does so in a fully computational manner. The step from a geometric theorem in Euclidean geometry to an algorithm using it has never been smaller, since there is now a direct correspondence between object names and their computational implementation.

5.7 Euclidean geometry using geometric algebra

To illustrate the geometric concepts we have discussed, and to show how to use the geometric algebra in computations, we conclude this section by illustrating three geometry theorems: Napoleon's theorem, Pappus' theorem, and Morley's triangle[9]. For all three of these theorems, we will work in the homogeneous model of two space with `e3` as the homogeneous coordinate.

5.7.1 Napoleon's theorem

Napoleon's theorem (attributed to Napoleon Bonaparte) states that if you have a triangle and you construct three equilateral triangles along its sides, then the centers of these equilateral triangles form another equilateral triangle. To begin, let's create and draw a triangle:

```
>> %GAblock(14)
>> % NAPOLEON'S THEOREM
>> clf;
>> P1 = e3; P2 = e3+e1; P3 = e3+e2;
>> DrawPolyline({P1,P2,P3,P1}, 'k'); GAview([0,90]);
```

[9]These and other interesting geometry problems can be found in The Penguin Guide of Curious and Interesting Geometry [14]

We now want to create the equilateral triangles. We already have two of the vertices of each triangle (they are two of the vertices of △P1 P2 P3); we only need to construct the third vertex to complete each equilateral triangle.

One way to construct this third vertex is to take an edge of our original triangle and rotate it by 60 degrees. We can make such a rotation operator by first constructing the rotation plane `i=e1^e2`, and then constructing our rotation operator taking `gexp` of `i` times the angle by which we wish to rotate (remember from section 3.4 that an angle is best viewed as a bivector!)::

```
>> %...
>> i = e1^e2; RR = gexp(i*2*pi/3);
```

We can now compute and draw the other two sides of the equilateral triangles by rotating the vector along each side of △P1 P2 P3 and adding it to one corner of the edge:

```
>> %...
>> S12 = (P1-P2)*RR+P1; DrawPolyline({P1,S12,P2}, 'b');
>> S23 = (P2-P3)*RR+P2; DrawPolyline({P2,S23,P3}, 'b');
>> S31 = (P3-P1)*RR+P3; DrawPolyline({P3,S31,P1}, 'b');
```

To complete the illustration of Napoleon's theorem, we need to compute the centroids of the three equilateral triangles and draw the triangle connecting them:

```
>> %...
>> C1 = (P1+S12+P2)/3;
>> C2 = (P2+S23+P3)/3;
>> C3 = (P3+S31+P1)/3;
>> DrawPolyline({C1,C2,C3,C1},'r');
```

We have written a routine, `napoleon`, to illustrate this theorem. To use the routine, add the `gable/Apps` directory to your Matlab path with the `addpath` command, and type `help napoleon` to see the calling sequence and a few examples of calls.

5.7.2 Pappus' theorem

Take any two lines and three points on each line (`P1 P2 P3` and `Q1 Q2 Q3`). Cross-join the points (i.e., build the line segments `P1Q2`, `P1Q3`, `P2Q1`, `P2Q3`, `P3Q1`, and `P3Q2`) and compute the intersection of the three cross-joined pairs of segments (i.e., intersect `P1Q2` with `P2Q1`, `P2Q3` with `P3Q2`, and `P3Q1` with `P1Q3`). Then Pappus' theorem states that the three points of intersection will be collinear.

To illustrate this theorem, we first need to construct six points and draw the relevant line segments:

```
>> %GAblock(15)
>> %PAPPUS'S THEOREM
>> clf
>> P1 = e3+e1; P2 = e3+2*e1; P3 = e3+4*e1;
>> Q1 = e3+e2; Q2 = e3+e1+2*e2; Q3 = e3+2*e1+3*e2;
>> DrawPolyline({P1,P3},'r'); DrawPolyline({Q1,Q3},'r');
>> DrawPolyline({P1,Q2},'k'); DrawPolyline({P1,Q3},'k');
>> DrawPolyline({P2,Q1},'k'); DrawPolyline({P2,Q3},'k');
>> DrawPolyline({P3,Q1},'k'); DrawPolyline({P3,Q2},'k');
>> GAview([0,90]);
```

Next we want to compute the intersection of corresponding line segments. As a first step, we need to compute each line segment (which we can do by joining two points on the segment) and as a second step we need to intersect pairs of line segments (which we do by computing the meet of the two segments). Note that the meet will give us a homogeneous point, and we will need to normalize its coordinates to put the point back in the homogeneous plane:

```
>> %...
>> H3 = meet(join(P1,Q2),join(P2,Q1)); A3 = H3/inner(H3,e3);
>> H2 = meet(join(P1,Q3),join(P3,Q1)); A2 = H2/inner(H2,e3);
>> H1 = meet(join(P2,Q3),join(P3,Q2)); A1 = H1/inner(H1,e3);
>> DrawHomogeneous(e3,H1,'n','g');
>> DrawHomogeneous(e3,H2,'n','g');
>> DrawHomogeneous(e3,H3,'n','g');
>> DrawPolyline({A1,A3},'b')
```

We have written a routine, `pappus`, to illustrate this theorem. To use the routine, add the `gable/Apps` directory to your Matlab path with the `addpath` command, and type `help pappus` to see the calling sequence and a few examples of calls.

5.7.3 Morley's triangle

Morley's triangle is constructed by taking any triangle and trisecting the angles. It states that the intersections of adjacent trisectors form an equilateral triangle. To illustrate this theorem, we need to trisect an angle. We will build the trisector of an angle by first constructing the rotation operator mapping one side to the other, taking the logarithm of this operator, which gives us the "bivector angle" of rotation. We divide this by three and exponentiate to get the desired rotation.

```
>> %GAblock(16)
>> %MORLEY'S TRIANGLE
>> clf;
>> P1 = e3; P2 = e3+e1; P3 = e3+e2;
>> DrawPolyline({P1,P2,P3,P1}, 'k'); GAview([0,90]);
>> R1 = (P3-P1)*(P2-P1); R13 = gexp(sLog(R1)/3);
>> R2 = (P1-P2)*(P3-P2); R23 = gexp(sLog(R2)/3);
>> R3 = (P2-P3)*(P1-P3); R33 = gexp(sLog(R3)/3);
```

We can now apply each rotation operator to the edges adjacent to the angle; by applying the operator on different sides, we get rotations of opposite sign. While these operators scale as well as rotate, they are sufficient for our purposes. By taking the outer product of each vector with the vertex at which it is rooted, we construct the desired trisecting line.

```
>> %...
>> L12 = P1^(R13*(P2-P1)); L13 = P1^((P3-P1)*R13);
>> L23 = P2^(R23*(P3-P2)); L21 = P2^((P1-P2)*R23);
>> L31 = P3^(R33*(P1-P3)); L32 = P3^((P2-P3)*R33);
```

Now we intersect the lines using the meet operation, normalize and draw these intersections, and draw the trisecting segments and the equilateral triangle.

```
>> %...
>> H1 = meet(L12,L21); C1 = H1/inner(H1,e3); DrawHomogeneous(e3,H1,'n','y')
>> H2 = meet(L23,L32); C2 = H2/inner(H2,e3); DrawHomogeneous(e3,H2,'n','y')
>> H3 = meet(L31,L13); C3 = H3/inner(H3,e3); DrawHomogeneous(e3,H3,'n','y')
>> DrawPolyline({P1,C1,P2},'g')
>> DrawPolyline({P2,C2,P3},'g')
>> DrawPolyline({P3,C3,P1},'g')
>> DrawPolyline({C1,C2,C3,C1},'r')
```

We have written a routine, `morley`, to illustrate this theorem. To use the routine, add the `gable/Apps` directory to your Matlab path with the `addpath` command, and type `help morley` to see the calling sequence and a few examples of calls.

Exercises

After these illustrations of the geometry of triangles, we have an exercise to let you practice on your own. When you solve it, use the proper operators we have defined to make your code stable under degeneracies (so `join`, `meet`, `connection`, etcetera): even if all the vertices coincide it should still compute sensibly. Do *not* use conditional case-statements; that is old-fashioned thinking about geometry which can now be replaced.

1. Define three points **P**, **Q** and **R**, define and draw the triangle determined by them. (Hint: use `connection`, `DrawHomogeneous` and `DrawSimplex`.)
2. Now degenerate the triangle by bringing all three points on the same line, or making some coincide. Your program should still give sensible results. But: *no* `if` *statements allowed!*
3. On a non-degenerate triangle, compute and draw the altitudes (lines perpendicular to the opposite sides), and their intersection (hint: use `connection`). Test if they intersect in one point, and if they do, draw that. (Hint: if the three mutual intersection points are separate, they would span an area; so test for that.) You should get something that looks like Figure 11.

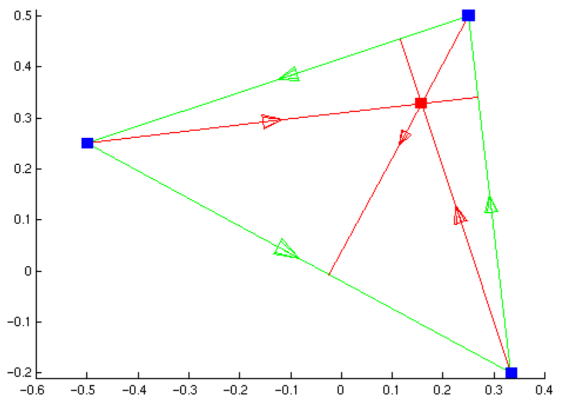

Figure 11: A triangle with its altitude lines.

4. Now draw the lines from the vertices to the midpoints of the opposite sides, and their possible intersection.
5. Now draw the perpendicular bisectors, using quantities from the previous two exercises.
6. Check computationally whether the intersection points of the three sets of lines we have defined are all on one line. (Hint: if they are not, they span an area – so compute the blade spanned by all three points.)
7. Write a GABLE routine illustrating Aubel's theorem, which says that for any quadrilateral, if you construct a square on each edge and form the two lines through centers of opposite squares, then these lines are perpendicular.
8. Write a GABLE routine illustrating Thebault's theorem, which says that if you construct square on each side of a parallelogram, then the centers of these squares are the corners of another square.
9. Write a GABLE routine to illustrate that the average of similar triangles yields a similar triangle. For the average to be similar, one triangle must be a rotated, scaled version of the other. Further, the averaging is performed by averaging like vertices.

   Your routine should take as arguments a triangle, an angle, and a scale factor. Draw the triangle, then scale and rotate it to produce a second triangle (which you should also draw). Then average like vertices and draw the resulting triangle.

   The theorem also holds for any weights where the weights sum to 1. For example, instead of adding 1/2 of one vertex to 1/2 of the like vertex of the other triangle, you could add 1/3 of one vertex and 2/3 of the other vertex (but be sure to use the same combination for all three pairs of vertices!). You may wish to extend your routine to take an optional argument of weights to apply.

6 Is this all there is?

Our tutorial stops here. We have familiarized you with the basic operators that geometric algebra offers, and some of their geometrical significance. The intuition acquired should enable you to read some of the other texts which are available, and get you into the right mode of thinking in geometrical concepts.

There are some things we left out of the tutorial. Since they are rather elementary, you may run into them soon after trying other texts, so a few words may be appropriate.

- *signatures*
  3-dimensional space can be given different signatures, so that the unit vectors do not square to 1, but to another scalar such as -1. We have chosen not to confuse the novice with the strange properties of non-Euclidean spaces, but GABLE can handle this: use `GAsignature`.
- *inner products*
  There are several inner products around; the one we have chosen is by no means standard, though it has the most straightforward geometric interpretation. The most common in

texts is the one introduced by Hestenes, which we have coded as `innerH`. You can play around with the other inner products through `GAitype`.

- *higher dimensions*
  At the moment, our implementation is *only* for 3-dimensional spaces. We have seen that a treatment of offset subspaces in Euclidean 3-space requires a 4-dimensional geometric algebra, so GABLE cannot be used for that at present. We will extend it in the near future. For now, you may want to use the Maple program of [5]. You can also use that for the conformal model of Euclidean space (which is 5-dimensional).
- *other models of Euclidean geometry; other geometries*
  In a fairly recent development (see for instance [9]), an improved embedding of Euclidean geometry into geometric algebra has been found. This is the *conformal model* of Euclidean space, and it is achieved by adding one more dimension to the homogeneous model, of negative signature. This dimension helps to represent the point at infinity (as our $\mathbf{e}$ represents the point at the origin). In this model, m-spheres are representable as elementary objects ($\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}$ is the circle through the points $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$), and the linear subspaces we have treated in the homogeneous model are then m-spheres containing the point at infinity. The principle is the same as in the homogeneous model: to do geometrically more sophisticated things, embed in a higher-dimensional Clifford algebra and use the products, `meet` and `join` there – that should provide all that is required: no new operators, only a new embedding. The basic operators still form the grammar of the geometry. In this same model, you can treat *projective geometry* and even *conformal geometry* more naturally. We may write a tutorial on that in the future.
- *differential geometry*
  A major subject we have *not* treated is *differential geometry*, and *geometric calculus*. The objects in geometric algebra can be differentiated, in new and powerful ways, and this is beginning to expand the scope of all fields using differential geometry. Matlab is less suited to show this flexibly. The Maple implementation of geometric algebra written in Cambridge [5] is a good tool to use when studying these new forms of differentiation (for instance combined with their course [3]).

Further reading

There is a growing body of literature on geometric algebra. Unfortunately much of the more readable writing is not very accessible, being found in specialized books rather than general journals. Little has been written with computer science in mind, since the initial applications have been to physics. No practical implementations in the form of libraries with algorithms yet exist (though there is a package for Maple [5] that can be used as a study-aid or for algorithm design, and we are currently developing a C++ package). We would recommend the following as natural follow-ups on this paper:

- The first part of the two part CG&A tutorial [4, 11] is mostly covered by this tutorial; the second part goes into more advanced topics.
- The introductory chapters of 'New Foundations of Classical Mechanics' [7].
- An introductory course intended for physicists [3].
- An application to a basic but involved geometry problem in computer vision, with a brief introduction into geometric algebra [8].
- Papers showing how linear algebra becomes enriched by viewing it as a part of geometric algebra [2, 6].

Read them in approximately this order. We are working on texts more specifically suited for a computer graphics audience; these may first appear as SIGGRAPH courses. The material from a course on Geometric Algebra that we co-presented at SIGGRAPH '01 can be found in the GABLE web page:

```
http://www.wins.uva.nl/~leo/GABLE/
http://www.cgl.uwaterloo.ca/~smann/GABLE/
```

This material is slides from our SIGGRAPH presentation, with the GABLE code we ran being available as "GAD"s (GA Demos); see the web page for details.

You can also find a large amount of additional literature from the usual sources. However, one warning about the literature: some people make no distinction between 'geometric algebra' and 'Clifford algebra' so you may want to check both; but you will find that the literature on the latter contains little basic geometry (it tends to be about spinors). Your best keyword for initial search is 'geometric algebra'. You will find that most of the literature in the field aims at an audience familiar with physics; yet the introductory chapters are often also readable to non-physicists. We trust that introductory material for the computer sciences will appear in the next few years.

7 Acknowledgments

We gratefully acknowledge the funding of Stephen Mann's sabbatical by the Dutch Organization for Scientific Research (NWO), and of Tim Bouma's summer stay by the University of Amsterdam.

A Matlab details

In this appendix, we discuss some Matlab implementation details that are relevant to using our geometric algebra package.

In many ways, Matlab eased the implementation of this geometric algebra. It was convenient to not have to write the matrix routines, etc. Also, Matlab objects and operator overloading nicely encapsulated many of the ideas we wanted to show while hiding many of the distracting details of the implementation.

On the other hand, using Matlab caused us to introduce several "warts" in GABLE. Some of these are straight-forward, such as the following:

- Matlab objects use standard arithmetic precedence for arithmetic on objects. This means that '^' has a higher precedence than '`*`', while for our system, we would want the two operators to have equal precedence. The standard arithmetic precedence may necessitate the use of parentheses at times.
- If a function name is the first entry on the command line, and the character following it is a space, then the rest of the line is treated as arguments to that function. We implemented `e1`, `e2`, `e3` as functions, which means you can not type

  ```
  >> e1 + e2
  ??? Error using ==> e1
  Too many input arguments.
  ```

  and must instead type

  ```
  >> e1+e2
  ans =
       e1 + e2
  ```

  However, any where else you may freely use spaces around `e1`, `e2`, `e3`. So for example,

  ```
  >> a = e1 + e2
  ans =
       e1 + e2
  ```

Other problems are more complex and subtle. This appendix describes some of these problems, how we addressed them, and what problems the user of GABLE may need to work around.

A.1 Global variables and naming conventions

We needed global variables to implement a few of our routines. We name our global variables starting with '`GA`', so in your own code, it is best to avoid using variables starting with '`GA`'.

The following were our naming convention for our functions:

- We use lower case function names for those functions that operator on `GA` objects. (An exception is made for a set of special routines as described in Section A.3; however, you probably won't need these special routines.)
- We use function names starting with '`Draw`' for the drawing routines, except for the basic drawing routine `draw`.
- Except for the frame routines, all other related routines that do not operate on `GA` objects begin with '`GA`'.

A.2 Graphics

The Matlab hold, axis, and rendering methods interacted poorly with our package. The following summarizes how we overrode Matlab's defaults and why:

- axis. To properly display the arrow heads of our vectors, we need equally scaled axes. If we have non-uniform scaling, the arrow heads look like open umbrellas, and further no longer appear perpendicular to the arrow body itself.

  To force equal axes, in the `draw` command we call `axis('equal')`. If you want non-uniform scaling of your axes, you will need to make a call to axis after your last call to `draw`.

  The problems are worse for `GAorbiter`, especially when drawing multiple subgraphs. `GAorbiter` sets `axis vis3d`, and rotates all subgraphs simultaneously. Further, `DrawOuter`, etc., call a routine that finds the min/max of all the subgraph axes (being careful of 2D plots) and sets all subplots to have the same axis.

- hold. We use the `patch` command to draw some of our objects and `plot3` command to draw other objects, and some objects are drawn with both. Matlab appears to call 'hold on' when patches are drawn, but does not make such a call when drawing lines. To make our drawing routines behave consistently for all geometric objects, we call `hold on` at various places internal to our routines.

- render. There are two rendering modes: 'painter' and 'zbuffer'. Unfortunately, the painter mode does not split overlapping polygons. Since all our bivectors overlap, drawing two bivectors in painter mode draws one on top of the other, which is incorrect. Thus, we set the default rendering mode to 'zbuffer', which correctly renders the bivectors.

  However, when saving the graphics screen to a PostScript file (such as for making figures), if you've rendered with the 'zbuffer' mode Matlab creates a bitmap image, which is huge and of low quality. If you're using the 'painter' rendering mode, Matlab will create a smaller, more reasonable PostScript file. If there is only one bivector in your image, this latter method is preferred. To allow for both, we wrote a `GArender` command. With no arguments, it returns the current rendering mode. With one argument, it sets the rendering mode to that argument.

A.3 Scalars

Scalar values caused us several problems. Matlab represents these as type `double`. However, a `GA` object is a matrix of eight scalars. When all but the first entry are zero, the `GA` object represents a scalar. For ease of use, our routines check their input for type double and automatically convert it to a `GA` scalar as needed.

We also had to decide what to do if the results of one of our procedures was a scalar. Our choices were to leave this as a `GA` object, which facilitates certain internal choices and simplifies some code, or to automatically convert scalars back and forth between `double` and `GA`. We chose the latter approach, as it fits more naturally with the rest of Matlab and seems more appropriate for the tutorial setting.

However, this automatic conversion interacts poorly with our overloading of the circumflex for the outer product. With automatic conversion of `GA` to `double`, if `A` and `B` in the expression `A^B` are both scalars, then both are converted to `double` and Matlab calls the scalar exponentiation function rather than our outer product function. Since the outer product of two scalars is actually the product of the two scalars, the result will be incorrect.

For simple testing of geometric algebra, the wedge product problem should not be a problem. However, if you are concerned, you can turn off automatic conversion of `GA` scalars to `double` by using the command `GAautoscalar`. Called with an argument of `0` turns off the auto-conversion, and called with an argument of `1` turns auto-conversion on.

`GAautoscalar` does not affect the automatic conversion of arguments to our routines from `double` to `GA` scalars. Since checking the type of the arguments takes time, we have a set of internal routines that assume their arguments are of type `GA` and always leave their results as type `GA`. If you prefer to do the conversion manually, you may use this alternative set of routines; note, however, that you will be unable to use the overloaded operators. Table 3 summarizes the correspondence between the two sets of routines.

To create a `GA` object from a scalar, use the `GAs` routine. To convert a `GA` object to a `double`, use the `double` routine.

Autoconvert	`GA` arguments
`*`	`GAproduct`
`+`	`GAplus`
`-`	`GAminus`
`^`	`GAouter`
`/`	`GAdivide`
`dual`	`GAdual`

Table 3: Correspondence between auto-conversion routines and `GA` routines

A.4 Numerics

Numerically, some routines (particularly the inverse routine) may create small error terms. For example, we might get a geometric algebra object that should be a vector, but has a small (on the order of 10^{-17}) bivector term. Several of our routines check to make sure that the arguments are blades. Thus, while the numerical error causes no particular computational problems, some routines will reject such geometric objects as not being blades. Thus, we wrote `gazv`, which sets all small terms of a geometric object to zero.

`gazv` will set to zero all terms of a GA that are smaller in absolute value than 1e-16, giving a warning when it does so. When developing code, it is a good idea to use `gazv` to overcome small numerical problems, and once the code is debugged switch to `grade` (since presumably you know the grade you want). Although you could use `grade` from the beginning, its use might hide some bugs that the system would otherwise automatically catch for you.

Since this software is meant for a tutorial, our `==` and `=` operators compare to within a numerical tolerance. Thus, vectors, etc., differing by only small amounts will be considered equal. If an exact equality is desired, one may use the `eeq` function. Any further testing will require extracting the coordinates using `inner`.

B Frames

In geometric algebra there is no particular basis for the space that is special: we can compute with any basis, and algebraically our formulas are independent of the coordinate system. Often, we do not even use the coordinates in our derivations. Computationally, however, we need to enter data and display it, which necessitated our use of a special frame. GABLE uses a default basis for $\mathcal{C}\ell_{3,0}$: `e1`, `e2`, `e3`.

However, in GABLE, we provide a routine to let you specify a different frame. To create a frame relative to vectors **u**, **v**, **w**, you should type

```
>> F = Frame('f', u,v,w, 'u', 'v', 'w');
```

The first argument is the frame name; the next three are the basis vectors; and the last three are the names of the basis vectors. The last three arguments are optional; if you omit them, then `Frame` will construct names for the basis vectors by appending '1', '2', and '3' to the frame name.

To print a geometric objective `g` relative to this frame, you type

```
>> charF(g,F)
```

You can use the `OFrame` command to change the default output routine so that all further output is printed relative to `F`. Both commands print the vector portion of a geometric object relative to the basis vectors we gave. And it is clear how to print the scalar, since that is independent of the frame. For bivectors, we had to choose a convention to make a bivector basis from the vector basis (this is of course not unique). We picked: $\mathbf{u} \wedge \mathbf{v}$, $\mathbf{v} \wedge \mathbf{w}$, and $\mathbf{w} \wedge \mathbf{u}$.

We can construct frames from any linearly independent vectors, including non-orthonormal frames. What follows is an example of the use of `Frame` and `OFrame`:

```
>> f1=e1+e2; f2=e2; f3=e3;
>> F = Frame('f',f1,f2,f3);
>> a=e1^(e1+e2+e3)
ans =
     e1^e2 + -1*e3^e1
```

```
>> OFrame(F);
>> a
ans =
     f1^f2 + -1*f2^f3 + -1*f3^f1
>> draw(a);
```

The first time we printed `a`, we were using the default frame. After setting the output frame using `OFrame`, `a` was printed using frame `f`. When you draw `a`, you will see it drawn relative to the standard frame `e1`, `e2`, `e3` (unless you had `axis off`). If you would like to see the `f1`, `f2`, `f3` basis vectors, you will have to draw them yourself.

To reset the output frame to be the default frame, call `OFrame([]);`.

Frames are a bit awkward because you can't write procedures that define variables outside their scope (except for global variables). Thus, when creating a frame, you have to pass both the vectors and the names of the vectors to the frame creation routine. For example, in the above example we carefully constructed the vectors `f1`, `f2`, and `f3` to match the names that the `Frame` command would use. This allows us to enter vectors relative to the frame as well as print relative to this frame. Thus, while we could call

```
>> G = Frame('g', e1+e2, e2+e3, e3+e1);
>> OFrame(G);
>> e1
ans =
     0.5*g1 + -0.5*g2 + 0.5*g3
```

we would be unable to enter values relative to frame basis elements `g1`, `g2`, `g3` because we haven't defined these Matlab variables.

Second, when we define the variables corresponding to the basis elements (as we did above for `f1`, `f2`, and `f3`), these are *local variables.* In particular, realize that if you pass the frame to a procedure, that procedure will be unable to create vectors relative to `f1`, `f2`, and `f3`, even if you passed the names of the vectors to this procedure. However, we do provide a means for extracting the basis vectors of a frame via the `FE` command. The command `FE(f,n)` returns the `n`th basis vector of frame `f`. Naturally, `n` should be an integer between 1 and 3.

We illustrate the use of `FE` by continuing the above example:

```
>> OFrame([]);
>> FE(F,1)
ans =
     e1 + e2
>> OFrame(G);
>> FE(F,1)
ans =
     g1
```

Thus, if you wish to pass a frame to a procedure and have it create vectors, etc., relative to that frame, the procedure could use the `FE` command to extract the basis vectors of the frame and proceed from there.

But let us emphasize that in geometric algebra, you use the actual frames only for the definition of the objects; none of their interactions or operations need ever use a particular frame representation; all are *coordinate-free*, in that sense.

C More about drawing routines

You may remember the three routines for illustrating the three products:

- `DrawOuter(A,B)`: Draw the outer product of A and B.
- `DrawInner(A,B)`: Draw the inner product of A and B.
- `DrawGP(A,B)`: Draw the geometric product of A and B.

Depending on the type of objects, all three draw routines will draw in one of two different modes. If neither object in the product is a mixed-grade multivector, then the drawing routine will bring up two subplots. In the left plot, you will see the first object drawn in blue and second object drawn in green. In the right plot, you will see the product of the two objects drawn in red.

Character	'r'	'g'	'b'	'c'	'm'	'y'	'k'	'w'
Color	red	green	blue	cyan	magenta	yellow	black	white

Table 4: Matlab colors

If either object is a mixed-grade multivector, then it is hard to visualize the two objects in a single plot. Therefore, the drawing routines will bring up three subplots. In the left plot, you will see the first object drawn in blue; in the middle plot you will see the second object drawn in green; and in the right plot, you will see the product drawn in red. If any of the three objects have a scalar part, a label will appear above the appropriate graph stating the value of the scalar.

After the graphs appear, you should use `GAorbiter` to get a better feel of the three-dimensional relationships between the objects.

C.1 Optional color arguments

Several of the drawing routines take an optional color argument. This color argument should be a Matlab color character, one of the colors given in Table 4.

The following list summarizes the effects of the color argument. The default colors are given in parentheses.

- `draw(a,c)`: draw the scalar (black), vector (blue), bivector (green with black border), trivector (red) or multivector (mixed) `a` in the color `c`.
- `DrawBivector(v1,v2,c)`: draw the parallelogram specified by `v1` (blue) and `v2` (green), with the interior shaded in `c` (yellow).
- `DrawPolyline(P,c)`: Draw the polyline specified by the cell array of vectors `P` in color `c` (blue).
- `DrawPolygon(P,c)`: Draw the polygon specified by the cell array of vectors `P` in color `c` (blue).
- `DrawSimplex(S,c1,c2)`: Draw the simplex specified by cell array `S`. The cell array `S` may have 1, 2, or 3 vectors in it. Each vector in `S` is drawn by calling `draw` with color `c1` (blue). Then draw the simplex formed by the tips of the vectors in `S` in color `c2` (yellow).
- `DrawHomogeneous(e,A,c1,c2)`:draw a simplex for the flat A in homogeneous representation. This draws the tangent blade at a position given by the perpendicular support vector. `e` is the special vector in homogeneous coordinates (typically e3); `A` is the homogeneous blade; `c1` is the color of the supporting vectors of the simplex; `c2` is the color of the simplex; `c1` and `c2` are optional, although `c1` must be specified if `c2` is used.

D Glossary

This glossary is meant to convey the geometric flavor of terms, rather than their exact definitions – for those the reader is referred to the appropriate sections.

- Bivector - A directed area element, a 2-dimensional direction. Formed by the outer product of two independent vectors, it determines a plane through the origin. (Section 2.2.2)
- Blade - a subspace through the origin; a k-blade is constructed as the outer product of k vectors. (Section 2.2.7)
- Dual - The orthogonal complement of a multivector (usually a blade); made through division by the unit volume. (Section 2.4.3)
- Geometric product - The product to scale, rotate and orthogonalize multivectors; it provides a geometric operator. (Section 2.4.1)
- Grade involution - take the k-blade $\mathbf{A}$ and multiply it by $(-1)^k$; for multivectors, do this for each of its blades. (Section 2.5)
- Homogeneous model - The embedding of n-dimensional Euclidean space in an $(n+1)$-dimensional geometric algebra. (Section 5.2)
- Inner product - the product used when orthogonal complements are involved. (Section 2.3)

- Inverse - The inverse under the geometric product; denoted by $\mathbf{A}^{-1}$ (as object) or by $/\mathbf{A}$ (as right-side operator). (Section 2.4.2)
- Multi-vector - a sum of blades, the most general element of geometric algebra. (Section 2.2.7)
- Outer product - The product used for spanning subspaces. (Section 2.2)
- Projection - The component of a blade totally contained in another blade; or the operation producing this. (Section 3.1)
- Pseudoscalar - The outer product of all the vectors in a basis for our vector space; the 'volume-element' of the space. (Section 2.2.3)
- Rejection - The component of a blade totally outside another blade; or the operation producing this. (Section 3.1)
- Spinor - a product of vectors, to be used as the operator $\mathbf{S}$ in the spinor product $\mathbf{S}\mathbf{x}\mathbf{S}^{-1}$. (Section 3.4.2)
- Trivector - A directed volume element. Formed by the outer product of three independent vectors. (Section 2.2.3)
- Vector - The basic 1-dimensional elements of a linear space spanning a geometric space. A basis for this vector space generates the linear space of the geometric algebra of the geometric space.
- Wedge product - Another name for the outer product. (Section 2.2)

References

[1] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic logic and mechanical theorem proving.* Academic Press, 1973.

[2] C Doran, A Lasenby, and S Gull. Linear algebra. In WE Baylis, editor, *Clifford (Geometric) Algebras with Applications in Physics, Mathematics and Engineering*, chapter 6. Birkhauser, 1996.

[3] Chris Doran and Anthony Lasenby. Physical applications of geometric algebra. Available at `http://www.mrao.cam.ac.uk/~`clifford/ptIIIcourse/, 1999.

[4] Leo Dorst and Stephen Mann. Geometric algebra: a computation framework for geometrical applications: Part i. *Computer Graphics and Applications*, 22(3):24–31, May/June 2002.

[5] Anthony Lasenby et al. GA package for Maple V.
Available at `http://www.mrao.cam.ac.uk/~clifford/software/GA/`, 1999.

[6] David Hestenes. The design of linear algebra and geometry. *Acta Applicandae Mathematicae*, 23:25–63, 1991.

[7] David Hestenes. *New Foundations for Classical Mechanics.* Reidel, 2nd edition, 2000.

[8] J. Lasenby, W J Fitzgerald, C J L Doran, and A N Lasenby. New geometric methods for computer vision. *International Journal of Computer Vision*, 36(3):191–213, 1998.

[9] Hongbo Li, David Hestenes, and Alyn Rockwood. Generalized homogeneous coordinates for computational geometry. Springer Series in Information Science. To be published 1999.

[10] Pertti Lounesto, Risto Mikkola, and Vesa Vierros. CLICAL user manual: Complex number, vector space and clifford algebra calculator for MS-DOS personal computers. Technical Report Institute of Mathematics Research Reports A248, Helsinki University of Technology, 1987.

[11] Stephen Mann and Leo Dorst. Geometric algebra: a computation framework for geometrical applications: Part ii. *Computer Graphics and Applications*, 22(4):58–67, July/August 2002.

[12] Stephen Mann, Leo Dorst, and Tim Bouma. The making of a geometric algebra package in Matlab. Technical Report CS-99-27, University of Waterloo, December 1999. Available as `ftp://cs-archive.uwaterloo.ca/cs-archive/CS-99-27/`.

[13] Stephen Mann, Leo Dorst, and Tim Bouma. The making of GABLE, a geometric algebra learning environment in Matlab. In E. Bayro-Corrochano and G. Sobczyk, editors, *Geometric Algebra with Applications in Science and Engineering*, pages 491–511. Birkhäuser, 2001.

[14] David Wells. *The Penguin Dictionary of Curious and Interesting Geometry.* Penguin, 1991.