3D Euclidean Geometry Through Conformal Geometric Algebra (a GAViewer tutorial)

Leo Dorst & Daniël Fontijne
Informatics Institute, University of Amsterdam, The Netherlands

Version 1.3, April 2005

Abstract

This tutorial introduces the conformal model of 3D Euclidean geometry, to date the most powerful way of using geometric algebra for Euclidean computations. We use GAViewer , a package for computation and visualization of the elements of this model, to establish the correspondence between geometric intuition and algebraic specification in this model.

Some basic knowledge of geometric algebra is assumed; it can be attained by doing our more basic GABLE + tutorial or reading some of the tutorial papers on our site. Both are available online at `http://www.science.uva.nl/ga/tutorials`.

1 Warming up to CGA

CGA (Conformal Geometric Algebra) is a very convenient model to do Euclidean geometry. It is built upon a representation of points and (dual) spheres. Internally, these are represented as vectors, but you do not need to know that to work with them. Also, no operations depend upon an origin, or need to be specified in terms of coordinates relative to that origin. In this sense, CGA is coordinate-free.

In this chapter, we get a feeling for its ease and possibilities, by playing around with the basic concepts. In chapter 2, we go into why this is possible and how it works – for that, you will need to know some geometric algebra (or Clifford algebra) – but for now you can get away with pattern matching. We have written a general tutorial for geometric algebra called GABLE, which you may find at `http://www.science.uva.nl/ga/tutorials/GABLE`.

Before starting the tutorial you will have to download our free program GAViewer from:

`http://www.science.uva.nl/ga/viewer`

You will also need a set of .g files that are contained in the `X_tutorial_files.zip` file you can find at:

http://www.science.uva.nl/ga/tutorials/CGA

Since GAViewer supports several models, we need to set ourselves up in the 3-D conformal geometric algebra, with a meaningful inner product. This is most simply done by loading in the entire directory of tutorial files for this tutorial and calling the function `init_c3ga()` This will load the necessary functions, and perform initialization. So, start GAViewer and under the File menu, select `File→Load .g directory` and point it to where you have put the .g files you downloaded. Then type the command:

```
init_c3ga()
```

at the console. If all went well, the console should now look something like

```
>> INIT: model is c3ga
INIT: inner product is left contraction
fontsize 24
>>
```

1.1 Points

First, we have a way to specify a (typical) point in our display. It is represented by the vector '`no`' (this name '`no`' is from 'Null vector modeling the Origin', as we'll explain later). On the console type:

```
a = no
a = 1.00*no
```

This creates a point. By Ctrl-RightMouse-Drag you can move it a bit (i.e. press the Ctrl key and the right mouse button at the same time while you are over the point, this selects it; then drag the mouse to move it - see also Figure 1). You can ask for the representation of this shifted point by typing '`a`', which will be something like:

```
a
a = 1.00*e1 + 0.50*e2 + 0.00*e3 + 1.00*no + 0.625*ni
```

So the point `a` has changed, and you see that in general there are 5 coefficients to the specification of a point. The coefficient of `no` is an overall weight, the coefficients of `e1`, `e2`, `e3` are its (weighted) position relative to `no`, and we will explain in section 2.1 that the coefficient of `ni` is proportional to half the modulus squared of the displacement vector. The $\{$`e1`, `e2`, `e3`$\}$ part is therefore how you would represent a point by its location vector in the regular representation of Euclidean space, the '`no`' part extends that to the representation of a point in the homogeneous model, and the '`ni`' part makes this into the new 'conformal' representation. This chapter will let you play around with such points to show that the conformal part really helps to make a very nice and compact description of Euclidean geometry.

Let us create some more points `b`, `c`, `d`:

	LeftMouse	MiddleMouse	RightMouse
	Rotate	Translate	Zoom
Ctrl	Select	Select	Select

Figure 1: *Mouse actions.*

```
b = c = d = no
```

and you can drag them to any place by Ctrl-RightMouse-Drag (repeaedly do Ctrl-RightMouse to cycle through different objects at the cursor location and note that the information bar at the bottom shows which one you have selected). You should tilt the viewing plane with LeftMouse-Drag to move the points to arbitrary 3D positions (if you don't, they will be in the same plane). We don't really care about their coordinates, in GA we never need them to describe the actual geometry, in the sense of the relationships and operations of objects.

Now you make the sphere 'spanned' by these four points simply by typing:

```
sphere = a^b^c^d
```

(Ignore the printed sphere coordinates, for now.) So this '$\wedge$' can span things. It is called the *outer product*. Using it on 3 points produces a circle:

```
circle = a^b^c
```

and doing it on 2 points makes a point pair:

```
ppair = a^b
```

(A point pair is blue, whereas individual points are red; but you can also show the points connected by typing `ppair` = `dm2`(`a` $\wedge$ `b`). Here `dm2` is 'draw method 2' for a point pair.) But it would really be nice to have all these objects change as we drag the points around, to convince ourselves that it always works. Let's do that by making the definitions 'dynamic':

```
dynamic{ sphere = a^b^c^d ,},
dynamic{ circle = a^b^c ,},
dynamic{ ppair  = a^b ,},
```

(Note the curly brackets, and the extra , just before the final } ! If you make a mistake, you have to remove the dynamic statements using `Dynamic`→`View Dynamic Statements` and remove it by ticking the box.) (You can use the 'up arrow' to go back to earlier statements, edit them, and re-enter them by pressing Enter.)

All these objects have a sense of orientation, and you can show this by selecting them, and then check 'draw orientation' in the Controls panel, which you can pop up by selecting it in the `View`→`Controls` menu. In this way, you should be able to see that the circle $a \wedge b \wedge c$ has the opposite orientation of $a \wedge c \wedge b$, but the same as $c \wedge a \wedge b$. Just drag the points around along the circle (make sure you look at it from straight above)

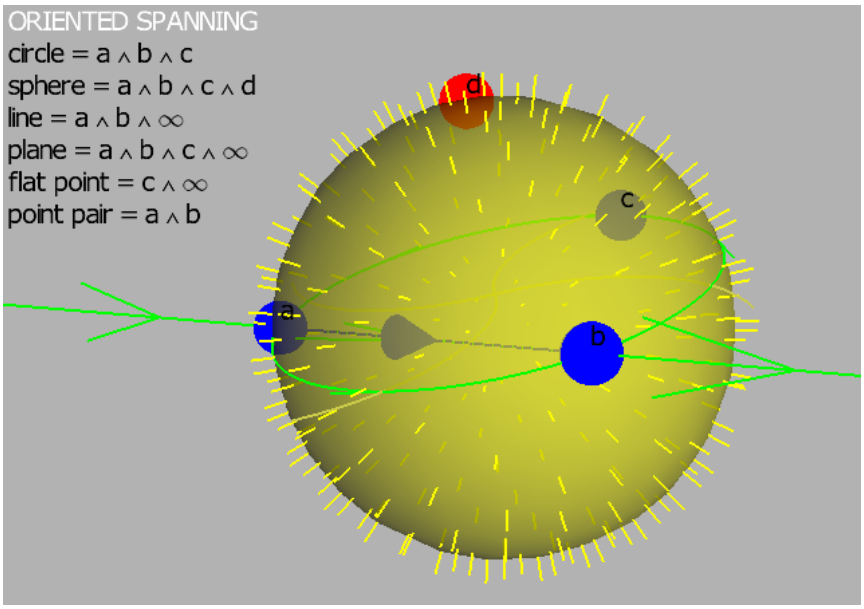


Figure 2: *Spanning rounds and flats in the conformal model.*

and see the orientation change as one point passes another. (If you find it hard to see the 'barbs' denoting the orientation, de-select 'draw weight', and they'll be drawn at a standard length.)

Algebraically, this means that the outer product $\wedge$ is anti-symmetric in any two of its arguments. It is also associative (so that defining it for 2 arguments extends to any number of arguments), so we do not need to write $(a \wedge b) \wedge c$ or $a \wedge (b \wedge c)$ – these are both equal to $a \wedge b \wedge c$.

Switch on the orientation display of the sphere, and note that it also changes orientation as you swap points. For the sphere, this happens when one of the points moves through the plane determined by the three others.

When you move the points around, you find that in some configurations, the circle almost becomes a straight line, and the sphere a plane, but that it is very hard to get that exact. There is, of course, also an explicit representation for these entities. Such 'flat' spaces go through the point at infinity, and in the conformal model, that is a regular point of the algebra. We have denoted it as '`ni`' (for 'Null vector modeling Infinity') – and we prefer to pronounce it at a slightly raised pitch.

So to form a line and a plane, do:

```
dynamic{ line = a^b^ni,},
dynamic{ plane = a^b^c^ni,},
```

As you see, the circle $C = a \wedge b \wedge c$ lies in the plane $C \wedge \mathtt{ni} = a \wedge b \wedge c \wedge \mathtt{ni}$, so we are beginning to glimpse a geometrically significant algebra. We call the elements in this

algebra *blades*: a circle is a 3-blade, a point a 1-blade, et cetera. The number of points used to make it is called the *grade* of the blade.

As an extra which you may skip, here is another little experiment, which we'll do in 2D space. To force GAViewer to a 2D usage, fix the camera orientation into a single pose. (Paradoxically, you can use a dynamic statement to make a constant, since it dynamically enforces the constant value, whatever happens.) When we then generate points and drag them, they will remain in the plane.

```
clearall();
dynamic{camori: camori=1; };   // this is a named dynamic statement
a = b = c = d = no,            // and drag them to wherever you want
dynamic{ circle = ori(a^b^c), }
dynamic{ sphere = ori(a^b^c^d), }
```

(The `ori` command shows the orientation.) Of course the sphere is now forced to be a plane (the plane we are observing from above). As you move `d` around, you see that the orientation of the plane switches just as `d` enters or leaves the circle (the front of the plane is solid yellow, the back of the plane shows the wireframe grid). Thus even in a plane, $\mathtt{a} \wedge \mathtt{b} \wedge \mathtt{c} \wedge \mathtt{d}$ is a useful quantity, its sign detects such a Euclidean 'inside' relation. Play around by making the circle $\mathtt{b} \wedge \mathtt{c} \wedge \mathtt{d}$, and convincing yourself that the signs are consistent: `d` enters $\mathtt{a} \wedge \mathtt{b} \wedge \mathtt{c}$ just as `a` leaves $\mathtt{b} \wedge \mathtt{c} \wedge \mathtt{d}$. Algebraically, it must be so, due to the symmetry properties of the outer product– and geometrically, it is indeed.

A compact demo illustrating all the above without the need to type is `DEMOspanning()`. But we felt that the hands-on feeling of typing the statements has some power of convincing you that this really is that easy.

You should now delete the dynamic statement for `camori` to continue the tutorial in full 3D (go to `Dynamic`→`View Dynamic Statements` and remove it, or type `cld(camori)` – you can remove or redefine such 'named' dynamic statements). Or remove all dynamic statements by typing `cld()`.

1.2 Complementation: the dual

Another important operation in the construction of elementary Euclidean objects is *dualization*. It is a 'complementary' representation of an object. It can be used to describe an object not so much specifying the points that are on it, but the points that are orthogonal to its complement.

If x is a point on a circle C (for instance constructed from other points as $C = a \wedge b \wedge c$), then we have $x \wedge C = 0$ since yet another point on the circle does not help to span a sphere or plane.

For the dual representation of C, i.e. $D \equiv \mathtt{dual}(C)$, the points on C are characterized as $x \cdot D = 0$, with $\cdot$ the geometrical inner product (basically the usual inner product extended to the more general spanned objects, as you may have learned from the GABLE + tutorial).

Of course we still think about a dual object in very much the same way as about the original object. So in GAViewer we plot it the same, but in a different color.

```
sphere = a^b^c^d,
dsphere = dual(sphere),
```

We draw direct spheres (or planes) yellow, and dual spheres (or planes) red.

Dualization makes some operations much more easy to specify, and is a powerful tool – we'll use it immediately in the next section. To 'undualize' you have to be a bit careful: depending on the dimensionality and metric of the geometric algebra model, there may be a change of sign involved to get back to the original object. In the CGA model for 3-dimensional Euclidean spaces, we have:

$$\texttt{dual}(\texttt{dual}(X)) = -X$$

so that 'undualization' of X in 3D CGA is the operation $-\texttt{dual}(X)$. For n-D Euclidean space, the sign involved is $(-1)^{n(n-1)/2}$, so you get this minus sign for the two cases of most interest $n = 2$ and $n = 3$.

1.3 Intersection

Apart from spanning using the $\wedge$, making object of higher dimensions, we can also intersect objects. The intersection of A and B is generally given by the operation

$$A \cap B = \texttt{dual}(B) \cdot A,$$

at least if they are 'in general position'. But it is much easier to remain in a dual representation, for

$$\texttt{dual}(A \cap B) = \texttt{dual}(B) \wedge \texttt{dual}(A),$$

so basically intersection is an outer product in a dual representation. We will get back to understanding this later in section 5, for now let's just play with it.

The above suggests that we first construct the objects we want using the outer product $\wedge$; then dualize them all using `dual()`; then use $\wedge$ intersect them in this dual representation. We may never choose to go back to the direct representation after this.

So make dual representations of our objects:

```
dynamic{ dsphere = dual(sphere) ;},
dynamic{ dcircle = dual(circle) ;},
dynamic{ dplane  = dual(plane) ;},
dynamic{ dline   = dual(line) ;},
```

The ; before the final curly bracket is an instruction not to draw the object constructed – if you want to see them, change it to a comma (but what you see for the circle may surprise you).

Now let's form another dual sphere and show the intersections with the objects defined. Ignore how this dual sphere is made for now, we just mention that it is at `no` and has unit radius.

```
dA = no-ni/2,
```

Now for the intersections. Before we make them, we simplify the situation a bit by putting the points in more standard positions relative to `no`. (To prevent mistakes, you could also run the whole demonstration by typing

```
DEMOintersect();
```

which also gives you some handy labels for the quantities to aid in dragging them - just do Ctrl-RightMouse-Drag on the label. It builds things up gradually, just keep pressing Enter while you see the `DEMOintersect` prompt.) To draw things properly, we undualize them, though for continued computations this is not really necessary.

```
a=pt(0), b=pt(e1), c=pt(e2), d=pt(e3),
dynamic{ dAsphere = -dual( dA^dsphere ) ,}
dynamic{ dAcircle = -dual( dA^dcircle ) ,}
dynamic{ dAplane  = -dual( dA^dplane ) ,}
```

(The function `pt`() creates a point that is the point `no` shifted over the vector specified.[1]) As you move the dual sphere `dA` around (Ctrl-RightMouse-Drag), you see the intersections change. Note that some are peculiar: apparently, two spheres always intersect in a circle, though this circle is not always on the spheres! Such strange intersecting circles are in fact imaginary (in the sense that their radius has negative square), and we have drawn them dashed to show that they are unusual.

Now move `dA` around again, in different planes (by tilting the view using LeftMouse-Drag), and note how intersections change. They always exist, for the system is 'closed' due to the inclusion of `ni`. If you're precise and capable of placing some elements such that they touch or are parallel, you'll see some strange objects which we'll discuss in chapter 2.

If you'd like to play some more, `DEMOincidence();` is similar.

1.4 Objects and their colors

We should now explain the colors of the various objects. For this, we need to look at those big coordinate expressions you saw come by – although we leave their full explanation to later.

The conformal model is capable of representing many different kinds of objects as elements in a geometric algebra – which is basically a linear space with a special product. Since it represents them in a linear space, you expect to be able to express everything in terms of a basis. We have seen that for points this basis is {`no,e1,e2,e3,ni`}.

Due to the regular structure of geometric algebra, the basis for the higher order objects (which are made as outer products) are just formed by combination of the basis elements using the outer products, in a completely regular manner. So for spheres, you would expect a basis containing $\{\mathtt{e1} \wedge \mathtt{e2} \wedge \mathtt{e3} \wedge \mathtt{no}, \mathtt{e1} \wedge \mathtt{e2} \wedge \mathtt{e3} \wedge \mathtt{ni}, \mathtt{e1} \wedge \mathtt{e2} \wedge \mathtt{no} \wedge \mathtt{ni}, \mathtt{e1} \wedge \mathtt{e3} \wedge \mathtt{no} \wedge \mathtt{ni}, \mathtt{e2} \wedge \mathtt{e3} \wedge \mathtt{no} \wedge \mathtt{ni}\}$. And indeed, typing

[1] Our `init.g` replaced its official name `c3ga_point`() – you may have to use that full name in your own implementations, or write your own `init.g`.

```
sphere
```

you see that it is expressed on such a basis. The basis for a dual sphere is just the dual of this basis, i.e. its complement. And indeed, typing

```
dsphere
```

we see that it is expressed on the basis $\{\mathtt{no}, \mathtt{e1}, \mathtt{e2}, \mathtt{e3}, \mathtt{ni}\}$. Look at the coordinates of

```
dual(e1^e2^e3^no},
```

and other combinations like it, to see that this basis is indeed dual to the earlier one. But $\{\mathtt{no}, \mathtt{e1}, \mathtt{e2}, \mathtt{e3}, \mathtt{ni}\}$ is just the same basis we used for points! Looking at that in another way, *points are just dual spheres with radius 0.* It therefore makes sense that both have the same color.

We have chosen the following colors to denote the 'grade' of an object (i.e. how many representation vectors were used to make it):

```
BLACK - 0 - these are scalars, not plotted
RED   - 1 - points and dual spheres, tangent & free vectors
BLUE  - 2 - point pairs, 'flat points', dual circles, tangent & free bivectors
GREEN - 3 - circles, lines, dual point pairs, tangent & free trivectors
YELLOW- 4 - spheres, planes
WHITE - 5 - pseudoscalars, not drawn

STIPPLE is used to denote that an object is imaginary, or that it is 'free'
         (i.e. translation invariant, see section 2.3).
```

If you need a reminder of these, type `DEMOgradecolors();`.

As you see, there are some objects we have not met yet – we'll treat them below in detail, but to lift some of their mystery, a 'tangent bivector' is the common element of two touching spheres, a 'free vector' is a 1-dimensional direction without a location, etc.

While we are on the subject of visualization, the slider for 'alpha' in the control panel of the interface allows you to change the opacity of the object. You can do that on the command line via

```
Y = alpha(X, 0.2)
```

but we hope to have chosen sufficiently sensible settings for the standard objects. We picked a see-through quality for planes and spheres so they don't clutter your view too much. The disadvantage of this is that OpenGL may not visualize the intersections between such objects (we plan to improve this), but if you really need those you can (and should) generate and draw them simply using the geometric algebra intersection operation of the previous section.

2 Elementary objects

We can now unveil how CGA works, by going through some of the details of the representation. To keep the explanation down to earth, we will occasionally refer to a coordinate representation. Although coordinates are not required to specify the operations of geometric algebra, they are of course still useful to specify its objects. We also give all of the kinds of objects that appear when we combine the span and dual operations (i.e. all intersections of spanned objects). After this section, you will be able to define lines, planes, etcetera simply, and with the precise locational and directional properties you desire.

2.1 Rounds and flats

We start with a point. The prototypical point is '`no`', the point at the (arbitrary) origin of our 3D space. Let us again pay attention to the representation of points. Make `a = no` and move it around with Ctrl-RightMouse-Drag to a new location. Then ask for its new representation, which will be something like:

```
  a
a = 1.00*e1 + 0.50*e2 + 0.00*e3 + 1.00*no + 0.625*ni
```

The general expression of a point at a location relative to `no` given by position vector the $\mathbf{p}$ (specified on the Euclidean basis $\{\mathtt{e1}, \mathtt{e2}, \mathtt{e3}\}$ is:

$$p = \alpha\, \mathtt{pt}(\mathbf{p}) \equiv \alpha\, (\, \mathtt{no} + \mathbf{p} + \tfrac{1}{2}(\mathbf{p}\cdot\mathbf{p})\, \mathtt{ni}\,)$$

(where $\mathbf{p}\cdot\mathbf{p}$ is of course the squared norm of $\mathbf{p}$, a scalar, and α is a scalar). Thus the function `pt( )` maps a Euclidean vector to the CGA representation of a point at that relative location. For instance:

```
p = pt(e1 + 2 e2)
```

These points are the basic elements of CGA. The inner product of CGA has been defined so that the basis elements of $\{\mathtt{e1}, \mathtt{e2}, \mathtt{e3}, \mathtt{no}, \mathtt{ni}\}$ have the following inner products:

	no	e1	e2	e3	ni
no	0	0	0	0	-1
e1	0	1	0	0	0
e2	0	0	1	0	0
e3	0	0	0	1	0
ni	-1	0	0	0	0

Note the special nature of `no` and `ni`: they are *null vectors* (i.e. their norm is zero), but in a sense each other's negative inverse under the inner product (since $\mathtt{no}\cdot\mathtt{ni} = -1$).

Why it has been set up in this way you find out when you compute the inner product between two normalized point representatives (for which the weight α equals 1):

$$\begin{aligned}\mathtt{pt}(\mathbf{p}) \cdot \mathtt{pt}(\mathbf{q}) &= (\mathtt{no} + \mathbf{p} + \tfrac{1}{2}\mathbf{p}^2\mathtt{ni}) \cdot (\mathtt{no} + \mathbf{q} + \tfrac{1}{2}\mathbf{q}^2\mathtt{ni}) \\ &= -\tfrac{1}{2}(\mathbf{p} - \mathbf{q}) \cdot (\mathbf{p} - \mathbf{q}) \\ &\equiv -\tfrac{1}{2}\, d_E^2(\mathtt{pt}(\mathbf{p}), \mathtt{pt}(\mathbf{q}))\end{aligned}$$

The inner product of two normalized points gives the square of the Euclidean distance! Since normalization is achieved simply through division: $\mathtt{pt}(\mathbf{p}) \rightarrow \mathtt{pt}(\mathbf{p})/(-\mathtt{ni} \cdot \mathtt{pt}(\mathbf{p}))$, we have for the vector representatives p and q of two points $\mathcal{P}$ and $\mathcal{Q}$:

$$\frac{p}{-\mathtt{ni} \cdot p} \cdot \frac{q}{-\mathtt{ni} \cdot q} = -\tfrac{1}{2}\, d_E^2(\mathcal{P}, \mathcal{Q})$$

The Euclidean distance measure is therefore deeply embedded into the algebra, and this means that all algebraic constructions incorporate it. (In the more classical approaches, we have to impose it explicitly by more cumbersome constructions.) Note that point representatives are null vectors:

$$\mathtt{pt}(\mathbf{x}) \cdot \mathtt{pt}(\mathbf{x}) = 0, \quad \text{for any } \mathbf{x}$$

All this makes it very easy to represent objects like planes and spheres, especially dually. We have seen that D 'dually represents' a point set iff the equation $x \cdot D = 0$ is true for precisely the points x in the set. So let us compute the midplane between two points a and b. A point x is on the midplane if its distance to either is the same. Using the inner product we simply express this as:

$$x \cdot a = x \cdot b.$$

Because of the properties of the inner product, this can be arranged to:

$$x \cdot (a - b) = 0.$$

It follows immediately that

$(a - b)$ *is the dual representation of the midplane between* a *and* b

A sphere with center c and radius ρ is not that much harder. We have as demand (for normalized points x and c):

$$x \cdot c = -\tfrac{1}{2}\rho^2$$

We want to rewrite this to something involving x explicitly. We do this using $\mathtt{ni} \cdot x = -1$, true for a normalized point x. This gives:

$$x \cdot (c - \tfrac{1}{2}\rho^2\, \mathtt{ni}) = 0,$$

so that

$s = c - \frac{1}{2}\rho^2\mathtt{ni}$ *is the dual representation of a sphere with center* c *and radius* ρ^2

Since s is a vector in CGA, we see that general vectors 'are' (weighted) dual spheres.

We will often make dual spheres at $\mathtt{no}$, which are simply

```
no - ni r r/2,
```

(Here we sneakily use the geometric product between `ni` and `r` and `r`, which is denoted by a space or by $*$ – we'll use it more later). So you often see us use the dual unit sphere at the origin: `no-ni/2`.

As you see, you can also make a sphere with a radius whose square is negative, for instance:

```
no + ni
```

We will call those 'imaginary spheres', and automatically stipple them.

The squared radius of a sphere can be found as the square of its dual (using the geometric product or the inner product):

$$s^2 = (c - \tfrac{1}{2}\rho^2 \mathtt{ni})^2 = \rho^2$$

(if you want to type this in, do something like `ds = pt(e1) - ni/20; rhosquared = ds ds,`). In this view, the points $\mathtt{pt}(\mathbf{x})$ are dual spheres with radius zero, since $\mathtt{pt}(\mathbf{p})^2 = 0$ for any $\mathbf{p}$. This will give us a nicely consistent semantics in section 5. The center of a sphere can be retrieved as:

$$c = -\tfrac{1}{2}\, s\, \mathtt{ni}\, s$$

which for now is just a magic formula (later you will recognize it as the reflection of the point at infinity into the sphere).

To get the actual sphere corresponding to $\mathtt{no} - \mathtt{ni}/2$, just *un*dualize it:

```
-dual(no-ni/2)
```

The difference in display is that a dual sphere is red (since it is an object of grade 1), whereas a direct sphere is yellow (being of grade 4).

A plane is also a grade 4 object, and in fact merely a sphere that also contains the point `ni`. Dually, it looks like

$$\mathbf{n} + d\,\mathtt{ni}$$

where $\mathbf{n}$ is the unit normal vector denoting its attitude, and d the distance to the origin. You may verify that this indeed represents the plane, by showing that:

$$\mathtt{pt}(\mathbf{x}) \cdot (\mathbf{n} + d\,\mathtt{ni}) = 0 \iff \mathbf{x} \cdot \mathbf{n} = d,$$

which is the usual 'Hesse normal equation' of a plane. Type

```
e1
```

and compare its color to

```
dual(e1)
```

A line can be made in various ways. You can do

```
a^b^ni,
```

showing that a line is determined by two points plus the point at infinity. You can also intersect two planes, for instance the planes dually represented by `e1` and `(e2+ni)`:

```
dual( e1 ^(e2+ni) )
```

Or you can specify it by a point and a direction. For a line through the point `b`, this is:

```
b^e1^ni
```

for a line in the e1-direction. Note that we need to put the special point `ni` in as well (any line passes through infinity).

Since a line is a grade 3 object, a dual line is of grade 2. We have incorporated some tests that recognize these particular grade 2 objects and draw them as the line they represent – but in blue, as befits a grade 2 object.

The combination of dual spheres and dual planes allows specification of dual circles rather easily, since the wedge is their dual intersection. So to specify a circle with radius 1 around the origin in the $\mathtt{e2} \wedge \mathtt{e3}$ plane, simply type:

```
dual( (no-ni/2)^e1 )
```

Note what happens when you leave out the dual:

```
(no-ni/2)^e1
```

This is an imaginary point pair, 'orthogonal' to the circle. We cannot automatically interpret this for you and draw it as a circle, since point pairs (even imaginary point pairs) are also legitimate objects. In fact, they are 1-dimensional spheres, the set of points of a line which have equal squared distance to a given point also on that line (the latter being the 'center' of the 1-dimensional sphere).

We have seen that $a \wedge b \wedge c \wedge \mathtt{ni}$ is a plane, $b \wedge c \wedge \mathtt{ni}$ is a line, and you may wonder what $c \wedge \mathtt{ni}$ is – probably a point, but wasn't simply '`c`' a point also?

```
c^ni
```

As you see, this is a grade 2 object (it is blue), and it looks like a point. We call it a 'flat point'. It is what you get when you intersect a plane and a line: these have *two* points in common, the regular intersection point plus the point at infinity. We can also say that they intersect in *one* flat point. It is hard to get this distinction between 'points' and 'flat points' well expressed in words, since we are not used to having to distinguish them in geometry, but they are very useful for unification as we'll see in chapter 5.

Recognizing them, a circle and a sphere *always* intersect in a point pair, which may be real (in the usual situation), imaginary (when they do 'not really' intersect) or a flat point (if they are a circle and a sphere through infinity, i.e. a line and a plane).

We will call planes, lines and flat points collectively *flats*, and spheres, circles and point pairs *rounds*. A flat is a round containing the point at infinity '`ni`'. As we will see later,

this means that it does not have a *size* in the way that spheres and circles do. Both flats and rounds have a *weight* (or if you prefer, a *density*), which you can see in the Controls panel by selecting the object (Ctrl-RightMouse). For some elements without size, we can show it directly (for instance, as the length of a tangent vector, see below).

2.2 Tangent blades

With all these objects, you might think we are complete: what more can there be when you intersect round and flat things except other round and flat things? However, there are several surprises. See what happens if you intersect a sphere with one of its tangent planes:

```
-dual(  (no-ni/2)^(e1+ni) )
```

Your display shows a disk, which the information bar in the Controls panel tells you is a 'tangent bivector'. It is what the sphere and the plane have in common at their point of intersection, which is slightly more than merely the point. You see it is grade 3 (since it is green), and you can think of it as an infinitesimal circle in a well-defined plane. We knew of no better way to draw it than this disk (but in the Control panel, you can select an alternative display method for these objects – in the command line mode this is done by `X = dm1(X)`, `X = dm2(X)`, etc.)

To make such a tangent object at the origin, type:

```
no^e1^e2
```

but beware: a tangent bivector at a point `c` is *not* made using the construction $\mathtt{c} \wedge \mathtt{e1} \wedge \mathtt{e2}$. (First drag the above object and type '`ans`' to enquire about the result. Then type something of the form $\mathtt{c} \wedge \mathtt{e1} \wedge \mathtt{e2}$ – you should see different kinds of terms on the basis.)[2]

Of course there are also tangent *vectors*, and you can make one at `no` by

```
no^e3
```

You can move it by dragging, and asking for '`ans`' you see that its coordinates become more complicated. Generalizing, a tangent scalar '3' at `no` is

```
no^3,
```

and as you see that is just a weighted point at `no` and

```
no^e1^e2^e3
```

is a tangent trivector at the point `no`, drawn as a sphere of volume 1.

[2]In fact, one can prove that it is $\mathtt{c} \wedge \mathtt{rcont}(\mathtt{e1} \wedge \mathtt{e2} \wedge \mathtt{ni}, \mathtt{c}) = -\mathtt{c} \wedge (\mathtt{c} \cdot (\mathtt{e1} \wedge \mathtt{e2} \wedge \mathtt{ni}))$ (where `rcont` is the right contraction). You could try to prove this yourself after section 6.2.

2.3 Free blades (attitudes)

And still, this does not exhaust the possibilities of basic object classes. Let us intersect two parallel planes:

```
-dual( e1^(e1+ni)),
```

and you find that the answer is

```
-e2^e3^ni
```

which is a form we have not seen before. It is a 2-dimensional direction element, which we draw stippled at the origin. Try to drag it away: you can't. This is a translation invariant element of CGA, and eminently suitable to be called an 'attitude': it has no position, only an attitude (or orientation, if you will).

We call this a *free bivector*, because it has no position. We could have drawn it anywhere, or as a 'bivector field'. We decided to draw it dashed at the origin and make it immovable, but this does *not* mean that it resides there – in fact, it resides nowhere at all, it merely has an attitude.[3] It certainly does not reside at the origin, which is arbitrary anyway – but we had to do something. Fortunately, you will find that these elements hardly occur usefully by themselves, but mostly as elements in the construction of more easily interpretable objects.

Similarly, the *free vector*

```
e1^ni
```

is drawn dashed at the origin; mind that it is different from `no`$\wedge$`e1` (which is drawn solid). It is a one-dimensional attitude, i.e. a direction vector. We now see that a line is in fact made as the outer product of a point and an attiude:

```
a^(e1^ni)
```

which corresponds to the idea of a location/direction pair, algebraically composed. You can drop the brackets since the outer product is associative, and change the order (giving a minus sign each time you swap two vectors) – that retrieves the representation we have seen before.

Similarly, a plane at a location `a` can be made using an attitude:

```
a^(e1^e2^ni)
```

Returning to the intersection of the planes that motivated our introduction of the attitudes, note what the dual of the intersection is:

```
dplane1 = e1-ni, dplane2 = e1+ni,
dynamic{ dint =  dplane1^dplane2,};
```

[3] to be consistent then, `ni` should be drawn as a stippled point at the origin. hmm, shall we?

Yes, it is the free vector denoting the separation of the planes in both magnitude and direction. We will see later that this can be used immediately to make the translation operator which transforms one plane into the other (as $T = \exp(\mathtt{dint}/2)$).

A small caveat: the common direction element of two lines `L` and `M` is not obtained as $-\mathtt{dual}(\mathtt{dual}(\mathtt{L}) \wedge \mathtt{dual}(\mathtt{M}))$ – that is zero. The reason is that the dualization should have been done relative to the common plane, not relative to the whole 3D space. In general, you should instead use $\mathtt{meet}(\mathtt{L}, \mathtt{M})$, which does this adaptation of the encompassing space automatically.

2.4 That's all

The above really does exhaust the objects that can be made by repeated application of outer product and dualization applied to vectors and hence as the 'closure' of the spans of points and their intersection. As you see, we have the classical repertoire of elements you use in linear algebra, and then some: spheres and tangents, free elements. All these are precisely related algebraically. None of these is what we call a 'vector' classically – that has in fact become an imprecise usage, since a 'normal vector' a 'direction vector' and a 'position vector' are all different (they react differently when the origin is moved, or when the space is transformed). We should reserve the term 'vector' for an element of the modeling algebra, rather than for the elements of geometry.

CGA enables precise definitions for each vector-based concept:

- 'normal vector' $\mathbf{n}$ (best seen as a dual plane $\mathbf{n}$, which is then automatically extended by translation to encode for its location, see below)
- 'direction vector' $\mathbf{v}$ (best seen as the attitude $\mathbf{v} \wedge \mathtt{ni}$)
- 'tangent vector' $\mathbf{t}$ (which is $\mathtt{no} \wedge \mathbf{t}$, translated to the desired location)
- 'position vector' $\mathbf{p}$ (this corresponds to $\mathtt{no} \wedge \mathbf{p} \wedge \mathtt{ni}$, a line element from `no` to $\mathtt{pt}(\mathbf{p})$, although this is freely shiftable along the line; it may be better to see a position vector as a direction vector to be used from `no`)

Each of these have well-defined properties and all are properly related within the unified framework.

3 A visual explanation

We can also show you more visually why this surprising characterization of rounds by blades works. In this chapter, we 'pop up' the `ni`-dimension graphically, by using the 3D CGA as a specification language for OpenGL commands, but we will necessarily show you only the CGA for a 2-dimensional Euclidean geometry. For us, this depiction helped to take quite a bit of the magic out (though not affecting the poetry of the procedure). But

on a first reading of this tutorial, you should probably go ahead to chapter 4, to get some more useful techniques. Switch to that now, if you want to.

We have seen that a point at $\mathtt{x}$ is represented as:

$$\mathtt{pt}(\mathtt{x}) = \mathtt{no} + \mathtt{x} + \tfrac{1}{2}\mathtt{x}^2\,\mathtt{ni}$$

In 2D, this requires a 4D space with a basis like $\{\mathtt{no}, \mathtt{e1}, \mathtt{e2}, \mathtt{ni}\}$. This would seem hard to visualize. However, the `no`-dimension works very much like the extra dimension in homogeneous coordinates: it allows you to talk about 'offset linear subspaces', linear subspaces that are shifted out of the origin (run `DEMOhomogeneous();` to remind yourself of this, and see Appendix A for an explanation if you need it. So because of the `no`-term, we are allowed to draw planes, lines, et cetera that do not need to go through the origin. If you accept that, we do not need to draw this dimension explicitly, we can just use this freedom and know that such things are blades because of the `no`-dimension.

The `ni`-dimension is new, and much more interesting. If we draw the Euclidean 2-space as the $\mathtt{e1} \wedge \mathtt{e2}$-plane, then there is apparently a *paraboloid* $\frac{1}{2}\mathtt{x}^2$ in the `ni`-direction that we should get to know better.[4]

Just execute the command

```
DEMOc2ga();
```

By hitting return, it will execute various stages of our visualization. For now, stop at the step where it says: `DEMOc2ga initialized >>` . (If you hit too far, just keep doing it till your prompt returns to the regular prompt, then restart `DEMOc2ga()`.) You see the 2-dimensional Euclidean space laid out in white, and the `ni`-paraboloid indicated vertically above it. The sliders in the bottom right of your window allow you to play with `pan` and `tilt` for better views.

Now we can play around. Let us first interpret a point `x` – actually, we use flat points like $\mathtt{x} \wedge \mathtt{ni}$. Hit return till you get to: `DEMOc2ga visualization of x >>` . As you move the red vector `x` around (by dragging its point), you see a yellowish plane move with it. This plane is the `dual(x)` (in the metric of 2D CGA): it consists of all the vectors perpendicular to the vector `x` (in this metric). (It doesn't *look* perpendicular, but that is because we are watching with Euclidean eyes.)

If $\mathtt{x}$ is on the paraboloid, this plane is the tangent to the paraboloid at that point. (Confirm this for yourself, if necessary by changing your viewpoint.) How would we express this? Well, in homogeneous coordinates `x` is on a plane `P` iff $\mathtt{x} \wedge \mathtt{P} = 0$. Or, if we have a dual representation of the plane, $\mathtt{p} = \mathtt{dual}(\mathtt{P})$, then `x` is in the plane iff $\mathtt{x} \cdot \mathtt{p} = 0$. You will

[4]When we do this, it gets a bit tricky and confusing that the `e3`-direction on our screen and in our 3D CGA is actually the `ni`-direction for the 2D CGA. And of course, we have to make it algebraically correct, for $\mathtt{e3}.\mathtt{e3} = 1$ whereas in 2D $\mathtt{ni}.\mathtt{ni} = 0$. A structural way of doing this is to recognize that the objects containing $\mathtt{ni}_3$ (the 3D `ni`) can be mapped to the lower algebra, as long as we interpret `e3` as $\mathtt{ni}_2$ (the 2D `ni`). The outer product and the duality, piped through this `c2ga`, need to be redefined using mapping operations. We define the necessary functions and set the scene with 2D Euclidean space (in white) and the paraboloid hovering over it. If you are interested in seeing how this is all set up, open the file `DEMOc2ga.g`.

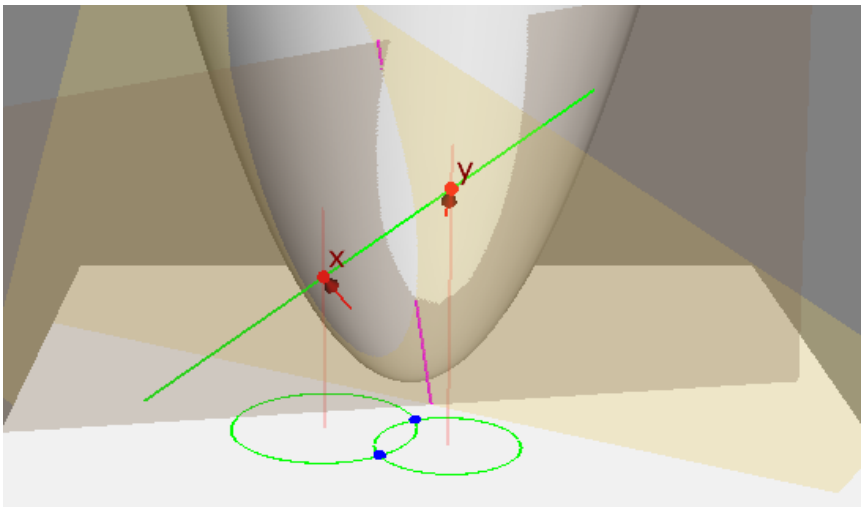


Figure 3: *Visualization of the intersection of circles in the conformal model.*

remember that the metric of CGA is set up in such a way that $\mathtt{x}\cdot\mathtt{x} = 0$, and the motivation for that was that a point represented by $\mathtt{x}$ has distance 0 to itself in the Euclidean metric. We now see that we can also read this as: if the vector $\mathtt{x}$ represents a Euclidean point, then $\mathtt{x}$ is on the plane dually represented by $\mathtt{x}$ in CGA. This is all consistent, for the parabola is given by the CGA metric, which in turn was designed to make the inner product of points be related to the (squared) Euclidean distance.

In the 2D CGA metric, the duality of a point to a plane works in a matter that you may discover by moving the point around: project the point $\mathtt{x}$ onto the parabola by a (vague red) line perpendicular to the Euclidean 2-space. The dual plane for $\mathtt{x}$ will be parallel to the tangent plane at this intersection point, but as far above the paraboloid as $\mathtt{x}$ is below it (or vice versa).

You see that the plane intersects the paraboloid in an ellipse (if you are at the proper side of it), and that we have drawn a circle in the 2D Euclidean plane as its projection. Apparently, there is a direct correspondence between the dual of a vector (the plane) and a Euclidean circle. But we know that there is. Let $\mathtt{c}$ be the representation of a Euclidean point – so $\mathtt{c}$ is on the paraboloid. Now subtract $\frac{1}{2}\rho^2\mathtt{ni}$ from $\mathtt{c}$, which gives the vector

$$\mathtt{s} = \mathtt{c} - \tfrac{1}{2}\rho^2\,\mathtt{ni}$$

This is the dual representation of a sphere (and in 2D, a sphere is a circle). But it is also a general vector of the form we have just moved around. If we enquire which actual Euclidean points are on this set, we have to enquire for which vectors $\mathtt{x}$, which satisfy $\mathtt{x}\cdot\mathtt{x} = 0$, the equation $\mathtt{x}\cdot\mathtt{s} = 0$ holds. The former demand is: $\mathtt{x}$ lies on the paraboloid, and the latter: $\mathtt{x}$ lies on the plane dual to $\mathtt{s}$. Together you can work it out as:

$$0 = \mathtt{x}\cdot(\mathtt{c} - \tfrac{1}{2}\rho^2\,\mathtt{ni}) = -\tfrac{1}{2}d_E^2(\mathtt{x},\mathtt{c}) + \tfrac{1}{2}\rho^2$$

(using $\mathtt{x} \cdot \mathtt{ni} = -1$, true for normalized points). So indeed these are the points `x` that have squared distance ρ^2 from `c`.

As you move the point `s` 'inside' the parabola, the dual plane lies outside it, and it seems there is no intersection. Actually, the intersection is imaginary, leading to a sphere with negative radius squared, but we hope that the interactive depiction provides the confidence that this is all completely regular. [5]

Now hit return again in the demo, to get to the prompt "`DEMOc2ga visualization of x and y >> `". The construction shows how the intersection of two spheres (circles in 2D) is done in CGA, and we explain it as follows.

A point pair is a 1D sphere. We know that the 2D CGA model would represent this as the outer product of two vectors $\mathtt{x} \wedge \mathtt{y}$, i.e. as a line in this homogeneous depiction. For two vectors representing points, this is easy enough: the vectors lie on the parabola, and so the intersection of the line with the parabola precisely retrieves the points. If the vectors `x` and `y` are off the paraboloid, they represent dual circles. Then their outer product dually represents the intersection of those circles. Undualizing should then provide the direct representation of this intersection, i.e. a point pair.

In the terminology of our visualization: make the planes corresponding to the vectors `x` and `y`, and intersect them to form a line ℓ. That is the representation of the intersection of the circles. To find the point pair, ask which points lie in the set ℓ; you do that by intersecting ℓ with the paraboloid. If it really intersects, the point pair is real, and otherwise imaginary (in a location that is rather counterintuitive but can be explained with some effort – look for hyperbolas, if you must...).

Move `x` and `y` around from the initial situation to get a feeling for how it is all connected. And you may enjoy typing the following to try a situation with two tangent circles (zoom, pan, tilt if necessary):

```
x = pt(-1.5 e2 + e3),
y = pt(-2 e2 + 1.5 e3),
```

Here is the take-home message of all this visualization:

> *In 2D CGA, 'intersecting circles' is identical to 'intersecting homogeneous planes' in 1 more dimension, which in yet 1 more dimension is identical to 'intersecting subspaces through the origin' – which is easy to do. So intersecting circles is easy – and so is intersecting general rounds in nD.*

Since spheres are important to Euclidean geometry (planes and lines et cetera are merely affine, not Euclidean), this is a relevant trick. We hope you now understand slightly better where those two extra dimensions come from.

[5] There is a small artifact of our depiction: if you would have `x` precisely *on* the paraboloid, the circle should degenerate to a 2D CGA point. But in our depiction (which fakes this using 3D CGA geometry, see previous footnote), it actually becomes a 3D CGA tangent bivector. Try it by defining $\mathtt{x} = \mathtt{pt}(\mathtt{e1} + \mathtt{e3}/2)$.

4 Orientations and weights

CGA automatically endows objects with a weight and an orientation, although sofar we have hidden those from the display. This section investigates how oriented objects interact. You can skip it at first reading and go immediately to the section on transformations.

4.1 Oriented intersections

We can show the orientation of the blades by using the `ori` command. Let us first take the line-like elements in the plane. We have set up a useful initialization routine `DEMOori2dinit();`, giving the oriented plane `no`$\wedge$`e1`$\wedge$`e2ni` and some labeled points. You could do all these demos through typing `DEMOori2d();`, but it is probably useful to see what is going on in some detail anyway.

```
DEMOori2dinit();
dynamic{line: line = ori(a^b^ni), }, label(line);
```

We have denoted the orientation of the plane by a windmill symbol (the way it would turn if you would blow on it is the orientation of the plane). The line is `a`$\wedge$`b`$\wedge$`ni` is oriented *from* `a` to `b`. The length of the barbs on the arrows along the line is an indication of its weight.

Let us make another line and intersect them. This intersection is best done after normalization of the lines, so that its weight fully depends on their relative stance.

```
dynamic{mine:  mine = magenta(ori(c^d^ni)), }, label(mine);
dynamic{ common = (normalize(line)/plane).normalize(mine), }
ctrl_range("weight" = 0, -1, 1);       // creates a slider
dynamic{ "weight" = (no^ni).common; };
```

The intersect point changes in size as you vary the lines (do this by dragging the red points defining them); we have denoted its weight on the scalar control panel on the side so that you can also see when it changes sign. The "`weight of` (`mine` & `line`)" is in fact the sine of the angle between `mine` and `line` – i.e. from the magenta `mine` to the green `line`. (This was our reason for normalizing the lines in the definition of `common`.)

Now let us study the oriented intersection of a circle and a line, starting with a fresh sketchpad.

```
DEMOori2dinit();
line = ori(normalize(no^e1^ni)),
dynamic{ circle = ori(normalize(a^b^c)),  },
dynamic{ meet_cl: meet_cl = dm2(ori((line/plane).normalize(circle))),},
```

Drag `a`, `b`, `c` or the `line`! The result of the intersection of the circle with the line (in that order!) is a point pair (which is possibly imaginary). This point pair has an orientation, denoted by the arrow. Note that the orientation is continuous when the point pair changes from being real to imaginary; in fact, in between it is a tangent vector, with the same orientation.

Q: Can you determine the rule governing the direction of this arrow for a real point pair?
A: In a plane with the same orientation as the circle, the orientation of the point pair is the order in which the line traverses them. In a plane with the opposite orientation, it is reversed. In a sense, we can see the orientation of the circle defining whether it bounds a blob (with the plane) or a hole (against the plane); then the order is always from the point that traveling along the line hits as 'background-to-foreground' to the other one.

What about two circles?

```
DEMOori2dinit();
dynamic{ circle1 = ori(a^b^c),  },
circle2   = yellow(ori(no^pt(e1+e2)^pt(e1-e2))),
dynamic{ common_cc = dm2(ori( (normalize(circle2)/plane).normalize(circle1) )),},
ctrl_range("sq_weight" = 0,0,1);
dynamic{ "sq_weight" = sq_weight(common_cc); };
```

It is the same rule as for the line and circle – but note that you should follow how the yellow circle hits the green, not the other way around, since that gives the opposite sign![6] The weight of the meet of the normalized circles is indicated in the slider in the scalar controls panel. Drag the yellow circle around to see how this weight changes: note that it becomes zero when the circles have the same center (so this is what it means for circles to be 'parallel'). (If you do the detailed computation, the attitude of the planar dual of the meet of two spheres is the separation of their centers; the magnitude of this vector is the 'weight' of the meet; this should not be confused with the 'size' of the meet which is the separation of the point pair. The geometric `meet` operation contains a lot of information!).

Extending this to lines, which are degenerate circles, we can use the same rule. Just view that as a limit process in which the 'first' intersection point is preserved (a in $a \wedge b$), as well as the tangents at which the two circles meet, and the other intersection point b moves to infinity. You should at least be able to convince yourselves that this gives the correct sign for the meet. So the 'sine of tangent turning' and 'moving inward or outward' are consistent descriptions, as long as the 'inside' of an oriented line is properly related to the orientation of the embedding space: for a counter-clockwise oriented space, the inside of a line is on its left (rotating with the space element would turn the direction of the line towards its inside).

Now we would like to study orientations in 3D. Let's start afresh, using `DEMOori3d();`. It first conjures up a line and a circle, constructed from four points. Convince yourself that the orientation of line and circle are as you expect, possibly moving them around (through manipulating `a`, `b`, `c`) to study other situations.

[6] In general, the meet of A (grade a) and B (grade b) within a common join space (grade j) differs in sign by a sign from its opposite: $A \cup B = (-1)^{(j-a)(j-b)} B \cup A$.

Pressing 'enter' introduces the next stage of the demo, where we have a plane, and study the intersection of line and plane. The orientation of the plane is again indicated by a windmill-like symbol. It is the same as the orientation of the circle.

Press 'enter' again to clear away the circle and pop up the meet of plane and line. The (normalized) meet of a line and a plane is a flat point (since it is blue), the outer product of the common finite point and the point at infinity. As you can see, we display a weight for the normalized meet (its apparent size changes), but we display it more quantitatively by the scalar control bar in the lower right. In a clear generalization to 3D of the 2D results, the weight of the normalized meet is the sine of the angle between the line and the plane.

The sign is given by the relative orientation of line and plane, within the total space. As you see, the space is oriented as $\mathtt{no}\wedge\mathtt{e1}\wedge\mathtt{e2}\wedge\mathtt{e3}\wedge\mathtt{ni}$. The formula we have implemented is the meet of the plane and the line (in that order). Follow the line in its orientation and cut the plane. Q: does the meet of a line and a plane have the same sign as the meet of a plane and a line, or not? A: see previous footnote.

The demonstration then continues (press 'enter' again) to illustrate the meet between two planes. It has a weight and an orientation. We here do the meet of the yellow plane with the pink plane (in that order!) and the orientation of the white line is related to that in a 'right handed' fashion since the pseudoscalar $\mathtt{I5} = \mathtt{no} \wedge \mathtt{e1} \wedge \mathtt{e2} \wedge \mathtt{e3} \wedge \mathtt{ni}$ is right handed. We also illustrate the weight of this normalized meet as the length of the blue tangent vector – it is again the sine of the angle between the two planes. You therefore can read off sign and magnitude of the normalized meet on the blue vector drawn along the line.

4.2 Parameters

In principle, computing parameters of the various kinds of blades is not too hard. For instance, the square of a normalized dual sphere gives you its radius squared, for

$$(\mathtt{no} - \tfrac{1}{2}\mathtt{ni}\,\rho^2)^2 = -\tfrac{1}{2}(\mathtt{no}\,\mathtt{ni} + \mathtt{ni}\,\mathtt{no})\,\rho^2 = \rho^2$$

For the direct representation of a round there may be extra signs (depending on its grade), and for a general formula you need to normalize first. Tangents have size 0, and for the flats and attitudes, size is not an issue, they don't really have any.

Instead, attitudes and flats only have a *weight*, an overall multiplicative factor relative to unity. The weights of $2\mathtt{e1}$, of $2\mathtt{e1}\wedge\mathtt{ni}$, of $2\mathtt{no}\wedge\mathtt{e1}\wedge\mathtt{ni}$ are all 2, but so is the weight of the tangent $2\mathtt{no}\wedge\mathtt{e1}$ and the dual round $2\mathtt{no}-\mathtt{ni}$. So tangents and rounds have weights too. In some cases, these weights have a traditional way of displaying: a vector of weight 2 can be depicted as having length 2, and a tangent bivector of weight 2 as an area element of 2 areal units. But we have not decided how to depict a sphere of weight 2, and if we would draw points (i.e. dual spheres of zero radius) as different sizes, they would easily become confused with spheres. So for some objects, you will just have to monitor the weight, for instance using the 'Controls' panel.

The set of functions defined in `conformal_blade_parameters.g` defines the various blade parameters as the functions. The actual computations are indicated in table 4.

class	attitude	flat	dual flat	tangent	round
form	$\infty\mathbf{E}$	$p \wedge (\infty\mathbf{E}) = \mathsf{T}_{\mathbf{p}}(o \wedge (\infty\mathbf{E}))$	$p \cdot (\infty\mathbf{E}) = \mathsf{T}_{\mathbf{p}}(\mathbf{E})$	$p \wedge (p \cdot (\infty\mathbf{E})) = \mathsf{T}_{\mathbf{p}}(o\mathbf{E})$	$v \wedge (v \cdot (\infty\mathbf{E})) = \mathsf{T}_{\mathbf{p}}((o + \alpha\infty)\mathbf{E})$
condition	$\infty \wedge X = 0$ $\infty \cdot X = 0$	$\infty \wedge X = 0$ $\infty \cdot X \neq 0$	$\infty \wedge X \neq 0$ $\infty \cdot X = 0$	$\infty \wedge X \neq 0$ $\infty \cdot X \neq 0$ $X^2 = 0$	$\infty \wedge X \neq 0$ $\infty \cdot X \neq 0$ $X^2 \neq 0$
attitude	X	$\infty \cdot X$	$\infty \wedge X$	$\infty \wedge (\infty \cdot X)$	$\infty \wedge (\infty \cdot X)$
location	none	$(q \cdot X)/X$	$(q \wedge X)/X$	$\frac{X}{\infty \cdot X}$	$\frac{X}{\infty \cdot X}$ or $\frac{1}{2}\frac{X\infty X}{(\infty \cdot X)^2}$
sq. weight	$(q \cdot \mathrm{att}(X))^2$	$(q \cdot \mathrm{att}(X))^2$	$(q \cdot \mathrm{att}(X))^2$	$(q \cdot \mathrm{att}(X))^2$	$(q \cdot \mathrm{att}(X))^2$
sq. size	none	none	none	0	$\alpha = -\frac{X\,\widehat{X}}{2(\infty \cdot X)^2}$
inverse	none	$p \wedge (\infty\mathbf{E}^{-1})$	$p \cdot (\infty\mathbf{E}^{-1})$	none	$\frac{1}{2}\mathsf{T}_{\mathbf{p}}((o/\alpha + \infty)\widehat{\mathbf{E}}^{-1})$

Figure 4: *All non-zero blades in the conformal model of Euclidean geometry, and their parameters. For a round, the squared size α equals radius squared, for a dual round it is minus the radius squared. Locations are denoted by dual spheres. The points q are probes to give locations closest to q, one can just use $o =$* `no`*. We denote* `ni` $= \infty$*, and $\widehat{X}$ is the grade inversion $(-1)^x X$ with $x = \mathrm{grade}\,(X)$, while $\widetilde{X}$ is the reversion $(-1)^{x(x-1)/2}X$. For more information see [1].*

The *location* of a blade could be the Euclidean coordinates of some relevant point. For a round, this is naturally the center, but for planes and lines such a point is not uniquely indicated in a coordinate free manner. We can either take the point closest to the origin (obviously not coordinate-free) or closest to some given point `q`. The formulas in the table actually produce a normalized dual sphere as the location; this is often enough, or you can take its Euclidean part as the Euclidean location vector (usable in a translation versor `tv()`), or compute the center by reflection `ni` into the round X of grade k by:

$$c = -\tfrac{1}{2}\frac{X\,\mathtt{ni}\,X}{(\mathtt{ni} \cdot X)^2}$$

All these class-dependent functions have been collected in `conformal_blade_parameters.g`, but in a rather coded way for fast usage. The most useful are:

```
function attitude(X)  -- gives attitude of X
function location(X)  -- location of X (dual round with correct center)
function sq_weight(X) -- squared weight of X
function sq_size(X)   -- squared size, radius is +/- 2 sq_size
```

A future version of this tutorial may contain some demonstrations and exercises with these parameters, for quantitative applications. (But maybe you can make some yourself and send them to us.)

5 Construction by containment and orthogonality

We have constructed elements by spanning, or by intersection of spanned quantities. There are many geometrical problems in which this is enough, but CGA also allows direct specification of objects with different partial data. When we explore the rules involved, we seem to uncover a new and compact language for Euclidean geometry. In this section, we give you a feeling for this very new subject, attempting to develop algebraic rules and geometric intuition in tandem.

Let us see how we could specify a sphere of which we know the center c and one point p on it. Recall that the representation of a dual sphere with center c was: $s = c - \frac{1}{2}\rho^2 \mathtt{ni}$. If we know a point p on it, we must have $p \cdot c = -\frac{1}{2}\rho^2$. Re-arranging terms (using the distributive law of inner product over outer product, as well as $p \cdot \mathtt{ni} = -1$) we find that the dual representation is:

$$c + (p \cdot c)\mathtt{ni} = p \cdot (c \wedge \mathtt{ni})$$

This is immediately converted into GAViewer commands:

```
clearall();
p = no, c = no, label(p);
dynamic{ s = p.(c^ni), }
```

Drag p and/or c to see the result. Notice that the sphere is drawn in red, since it is a dual sphere. Note also that $c \wedge \mathtt{ni}$ is a 'flat point' – we will get back to the geometrical intuition behind this equation below.

Key to the correspondence of geometrical intuition and algebraic expressions are two rules, involving 'being part of' and 'being perpendicular to'. Both can be given in direct form, and in dual form, and all four together provide our framework. We state them without proof. (In each of these expressions, the inner product is the default in our software: the contraction inner product.) We use the notation $\cdot^*$ to denote dualization, for easy reading of the formulas.

- *containment:* for vector $\mathbf{x}$ and blade $\mathbf{A}$ (with grade at least 1)

$$\mathbf{x} \in \mathbf{A} \quad \Longleftrightarrow \quad \mathbf{x} \wedge \mathbf{A} = 0 = \mathbf{x} \cdot \mathbf{A}^*$$

- *perpendicularity* for blade $\mathbf{A}$ and blade $\mathbf{B}$ (with grade $(A) \leq$ grade (B))

$$\mathbf{A} \perp \mathbf{B} \quad \Longleftrightarrow \quad \mathbf{A} \cdot \mathbf{B} = 0 = \mathbf{B}^* \cdot \mathbf{A}^*$$

Let us play with that. Suppose we have three spheres A, B, C, and are looking for the GA object X that is perpendicular to each. We therefore need to satisfy $X \cdot A = 0$, $X \cdot B = 0$, $X \cdot C = 0$. Dualizing this and introducing $a = A^*$, $b = B^*$, $c = C^*$, we get $X \wedge a = X \wedge b = X \wedge c = 0$. The simplest object satisfying this is:

$$X = a \wedge b \wedge c$$

Done. Let's show it.

```
clearall();
a = no-ni/4, b = no-ni/2, c = no-ni,
dynamic{ X= a^b^c,}
```

Drag `a`, `b`, `c` around, and be convinced.

The outcome is interesting. The *direct* representation of the object perpendicular to other objects is the outer product of their *duals*. Shrinking the spheres to points, you see that you get a circle through them. So in this interpretation, *points are small dual spheres*, and to 'pass through' a point means to cut its corresponding *direct* sphere perpendicularly. This neatly unifies the point description with the spheres in one consistent scheme. This is a subtle point, and it pays to pause here a moment to let it sink it and make it your own.

Now let us revisit the object $p \cdot (c \wedge \mathtt{ni})$. It was a dual sphere through the point p, with center c. Dualizing this, we see that it is the direct object

$$p \wedge (c \wedge \mathtt{ni})^*.$$

With what we have just learned we see that this indeed contains p, and that we can think of it as being perpendicular to the flat point $c \wedge \mathtt{ni}$. Since the result has to be the sphere, this suggests the intuitive picture of Figure 5: a flat point has 'hairs' extending to infinity, and our object cuts them orthogonally. These hairs therefore help to construct an object consisting of points equidistant to c. At the right of the figure, we see a similar explanation

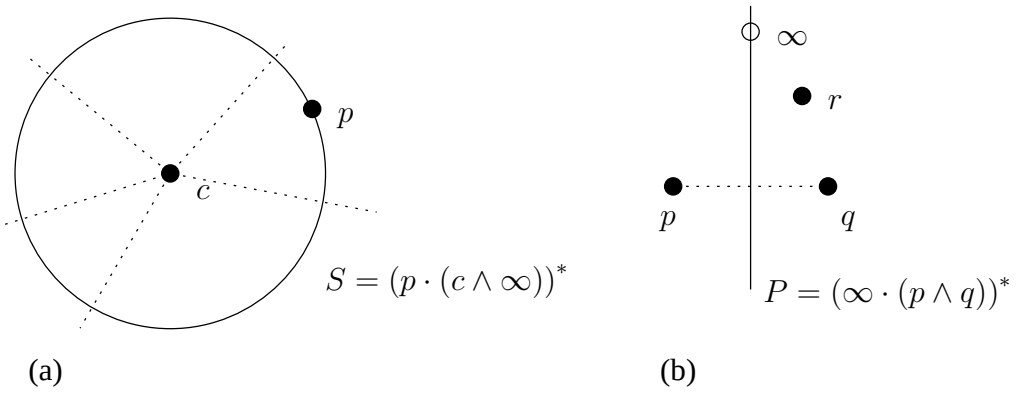


Figure 5: *The specification of dual spheres and planes, see text.*

for the construction of the midplane between two points p and q, which is $q - p$, or written multiplicatively and dualized:

$$(q - p)^* = (\mathtt{ni} \cdot (p \wedge q))^* = \mathtt{ni} \wedge (p \wedge q)^*.$$

Therefore the direct representation contains `ni` and cuts $p \wedge q$ orthogonally, a fair description of the midplane.

If we replace `ni` with a finite point r, we get $r \cdot (p \wedge q)$. Please explore its meaning yourself, and verify your insights using a small implementation.

With what we have learned, we can also interpret an object like

$$\mathtt{p} \wedge \mathtt{e1} \wedge \mathtt{ni}.$$

It contains the points `p` and `ni`, and should be orthogonal to $\mathtt{e1}^*$, which is the (`e2`∧`e3`)-plane through the origin. Obviously this is the line through `p` in the `e1` direction.

Now you can play with `DEMOortho();` and get some more intuition for the construction of such blades. It constructs the circle through p perpendicular to the planes dually characterized by `e1` and `e2`, you would simply type:

```
dp1 = e1,
dp2 = e2,
p = no,
dynamic{c = p^dp1^dp2,},
```

In the setup in the demo, this gives a 2-tangent, and as you move `p` a little you see that that is in a sense an infinitesimal circle. Then `DEMOortho();` proceeds to construct more elements (see its legenda in the upper left), which you should try to understand.

Let's get back to the incidence operation $A \cap B = -\mathtt{dual}(\mathtt{dual}B \wedge \mathtt{dual}A)$ from section 1.3. We now recognize the outcome as the dual of a blade that is the composed (by spanning) of elements perpendicular to both A and B. The result is therefore in both A and B.

We can use our new insights to show the relationship between the direct representation of a sphere as the outer product of four points $S = a \wedge b \wedge c \wedge d$, and the dual representation by a center m and a point a on it which is $s = a \cdot (m \wedge \mathtt{ni})$. This is illustrated in `DEMOspheres();`, which you may run in tandem with the algebraic explanation below.

We realize that the center should be the intersection of the midplanes of three points pairs. These midplanes are dually represented as $b - a$, $c - a$ and $d - a$, and the dual of their intersection is their outer product. However, this is not merely the center m, since `ni` is also on all planes. Therefore:

$$(m \wedge \infty)^* = \alpha\,(b - a) \wedge (c - a) \wedge (d - a)$$

(which α some proportionality constant) This helps use relate the two immediately. The dual representation gives $0 = x \cdot s$, and we dualize this and rearrange:

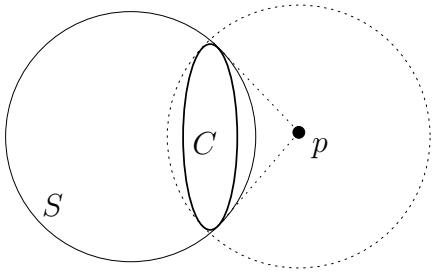
$$\begin{aligned} 0 &= (x \cdot (a \cdot (m \wedge \infty)))^* \\ &= x \wedge (a \cdot (m \wedge \infty))^* \\ &= x \wedge (a \wedge (m \wedge \infty)^*) \\ &= \alpha\; x \wedge (a \wedge (b - a) \wedge (c - a) \wedge (d - a)) \\ &= \alpha\; x \wedge (a \wedge b \wedge c \wedge d) \end{aligned}$$

This is of the form $0 = x \wedge S$, so we have found the direct representation. Done! And this also shows why the representation space for 3D Euclidean geometry is 5-dimensional: the dual of a vector is a 4-blade.

It is rather satisfying that such geometrically involved computations can be done so simply in CGA, without even introducing coordinates!

Puzzles

1. Give an expression for the contour C of a sphere S as seen from a point p. (Hint: try to characterize this construction by perpendicularity rather than tangency.)



2. Give the direct specification of a sphere S with a given tangent bivector B, going through a point p.

3. A point p is algebraically a null vector, i.e. it satisfies $p \cdot p = 0$. Interpret this geometrically in terms of perpendicularity.

4. Can you make the circle through the point p, passing through it in the direction `e1`, and orthogonal to the plane `e2`? (This is perhaps a bit hard at this point, but after the next chapter you should be able to do this.)

6 Transformations

The outer product, dual and inner product are not the only products in geometric algebra. In fact, they are all just special cases of the powerful 'geometric product'. We introduce its usage by treating transformations of objects. Definitions and explanations are similar to those in the GABLE + tutorial, here just enjoy how nice and general it all has become in the CGA context.

6.1 Reflections

An object can act as a reflector, to mirror any other object. This is done in a sandwiching operation, which uses the 'geometric product' and its inverse, the geometric division. If M is the mirror, then $M\,A/M$ is the reflection of an object A.

```
clearall(),
X = no+ni, label(X);
M = dual(e1+ni),
dynamic{mX: mX = M X/M,},
```

Even a sphere can be a mirror:

```
M2 = dual(no-ni/2),
dynamic{m2X: m2X = M2 X/M2,},
```

You see this best by moving the spheres close to each other, without having them intersect.

All these reflections work on any kind of object, for instance also on a line:

```
X = no^e2^ni,
```

Note that the reflection of the line into the sphere `M2` is a circle through its center. This is also known as an 'inversion' of the line, an studied in 'inversive geometry'.

Actually, the above equations, though structurally correct, do not quite reflect with the correct sign, so the orientation of objects gets messed up. You can inspect this for instance by

```
X = ori(no^(e1+e2+e3)^ni),
```

and by changing the dynamic statements to:

```
dynamic{mX: mX  = ori(M X/M),},
dynamic{m2X: m2X = ori(M2 X/M2),},
```

(You can make this change by typing this in on the console, the two 'named' dynamic statements `mX` and `m2X` are then replaced. Alternatively, you can edit those statements directly, select Dynamic→View Dynamic Statements.) You see that the signs of the reflections are wrong: the line is supposed to leave from its reflection in the opposite direction from which it entered. The full formula to reflect `X` in a blade `A` should be:

$$\mathbf{X} \quad \mapsto \quad \mathbf{A}\,\mathbf{X}/\mathbf{A}(-1)^{\mathrm{grade}(\mathbf{X})(1+\mathrm{grade}(\mathbf{A}))} = 0$$

(Here grade $(()\,\mathbf{X})$ is obviously the grade of X, the number of vectors used to form it in the outer product. In GAViewer you would type the right hand side as `A X/A pow(-1,(1+grade(A))grade(X))`.) We will derive it below. The formula can be used to detect whether an object $\mathbf{X}$ is contained within a blade $\mathbf{A}$:

$$\mathbf{X} \subseteq \mathbf{A} \quad \Leftrightarrow \quad \mathbf{X}\,\mathbf{A} - \mathbf{A}\,\mathbf{X}(-1)^{\mathrm{grade}(\mathbf{X})(1+\mathrm{grade}(\mathbf{A}))} = 0 \tag{1}$$

(One usually gets such signs upon changing the order of multiplication from the left multiply $\mathbf{X}\,\mathbf{A}$ to the right multiply $\mathbf{A}\,\mathbf{X}$.)

Here's the derivation of that sign in the containment test. The outer product for a vector $\mathbf{x}$ and a blade $\mathbf{A}$ (or even a general multivector) can be defined in terms of the geometric product as

$$\mathbf{x} \wedge \mathbf{A} = \tfrac{1}{2}(\mathbf{x}\,\mathbf{A} + (-1)^A \mathbf{A}\,\mathbf{x}),$$

where A is the grade of $\mathbf{A}$. So when $\mathbf{x}$ is in $\mathbf{A}$, so that $\mathbf{x} \wedge \mathbf{A} = 0$, we have the commutation relationship

$$\mathbf{x} \in \mathbf{A} \quad \Longleftrightarrow \quad \mathbf{x}\,\mathbf{A} = -(-1)^A \mathbf{A}\,\mathbf{x} = (-1)^{A+1} \mathbf{A}\,\mathbf{x}$$

If we now have a blade $\mathbf{X}$ of grade k, we can factor it into orthogonal components $\mathbf{X} = \mathbf{x}_k \, \mathbf{x}_{k-1} \cdots \mathbf{x}_1$. If $\mathbf{X}$ is to be a subblade of $\mathbf{A}$, each of these components should be in $\mathbf{A}$. Therefore we find:

$$\begin{aligned} \mathbf{X} \, \mathbf{A} &= \mathbf{x}_k \cdots \mathbf{x}_2 \, \mathbf{x}_1 \, \mathbf{A} \\ &= \mathbf{x}_k \cdots \mathbf{x}_2 \, ((-1)^{A+1} \mathbf{A} \, \mathbf{x}_1) \\ &= \cdots \\ &= (-1)^{k(A+1)} \mathbf{A} \mathbf{x}_k \cdots \mathbf{x}_2 \, \mathbf{x}_1 \\ &= (-1)^{k(A+1)} \mathbf{A} \mathbf{X} \end{aligned}$$

which shows clearly that the somewhat unexpected sign is a direct consequence of the k-fold outer product.

We remark that you should be careful to use these formulas exclusively on *direct* representations of the mirror $\mathbf{A}$, rather than on their duals. But it is easy to derive the formula using duals. Set $\mathbf{a} = \mathbf{A}/\mathbf{I}$, with $\mathbf{I}$ the pseudoscalar of $(n+2)$-dimensional CGA. Let $A = \text{grade}\,(\mathbf{A})$ and $a = \text{grade}\,(\mathbf{a}) = n + 2 - A$. Then

$$\mathbf{AX}/\mathbf{A}(-1)^{x(A+1)} = \mathbf{aIXI}^{-1}/\mathbf{a}(-1)^{x(A+1)} = \mathbf{aII}^{-1}\mathbf{X}/\mathbf{a}(-1)^{x(A+1)+x(n+3)} = \mathbf{aX}/\mathbf{a}(-1)^{xa}$$

This is the correct oriented reflection for a blade $\mathbf{X}$ in an object *dually* represented by the blade $\mathbf{a}$. From it, you can see what the containment test would be for such mixed representations.

Puzzle

1. Set up the reflection in a circle using a `dynamic` statement and play around with it. It may seem counterintuitive, until you realize that a circle can be represented as the dual of a geometric product of a plane (which plane?) and a sphere (which sphere?). Spell it out in that way with two more `dynamic` statements, and see if you understand the results now.

6.2 Translations

A free vector $\mathbf{v} \wedge \mathtt{ni} = \mathbf{v}\,\mathtt{ni}$ characterizes a translation over $\mathbf{v}$. The actual translation is the versor:

$$T = e^{-\mathbf{v}\,\mathtt{ni}/2} = 1 - \tfrac{1}{2}\,\mathbf{v}\,\mathtt{ni},$$

where the simplification on the right follows as the Taylor series of the argument in which all but the first two terms are zero. This versor is to be applied to an object X in the sandwiching product $X \;\mapsto\; T\,X/T$. So for instance:

```
V = 1 - e1 ni/2;
dynamic{ TA: TA = V A/V,},
A = no, label(A);
```

and as before, you can redefine `A` to be any of our elements. Play around with that; note that it makes no difference whether you use the objects or their duals. Especially also try a free attitude like $\texttt{A} = \texttt{e1} \wedge \texttt{e2} \wedge \texttt{ni}$, and notice that is translation invariant. We provide the shorthand

```
V = tv(t),
```

as the translation versor over the vector `t`.

It is hard to see which is the original, and which the translated version, so let us define a primitive 'trail' of subsequent applications of the versor, ever weaker. You have loaded the file `vtrail.g` when you loaded in the tutorial directory. Here is its definition:

```
batch vtrail(V,A,n)
// draw the trail of n actions of the versor V onto A
{
label(A);
A[0] = alpha(A,0.99),
for (i=1;i<n;i=i+1) { A[i] = alpha(vp(V,A[i-1]),1-i/n), }
}
```

where we use the slightly more efficient `vp`$(\cdot)$ for the versor product. You get lots of objects, many undraggable. Dragging `A` is most easy by selecting its label (Ctrl-RightMouse) and dragging that instead.

We use this in a dynamic statement to get an impression of the 'orbit' of a versor:

```
dynamic{ vtrail(V,A,5),}
```

(You'd better kill the now superfluous dynamic statement we defined before, using the Dynamic menu or by typing `cld(TA)`, or you'll get confused about the display of the once-translated version of `A`.)

Defining different types of `A` and dragging them around now gives a good feeling of what the translation versor does.

There is another way of looking at a translation, which will generalize in an interesting manner to the other rigid body motions: *a translation versor is the ratio of two flat points.* That two points determine a translation is of course obvious (classically, you would take their difference vector to characterize it), but it is nice that it is a ratio – we'll see that this generalizes. Let's just check this (assuming you still have `vtrail` set up, and `A`).

```
X = pt(e2)^ni, Y = pt(e2+e1/3)^ni,
dynamic{V = Y/X,},
```

The translation is over *twice* the distance of `X` and `Y`. The one making X into Y is the square root of this rotor V. For fun, try a different `A` such as $\texttt{A} = \texttt{no} \wedge \texttt{e1}$.

Puzzle

1. A translation can also be viewed as two reflections in parallel planes. Do this in GA, and show algebraically that the translation is over twice the plane separation, in the direction of their normal.

6.3 Rotations

In 3D, a rotation axis ℓ defines a rotation. An axis through the origin has a dual that is a purely Euclidean bivector $\mathbf{B} \equiv \texttt{dual}(\ell)$, so that also defines a rotation – it turns out that this generalizes to n-D, so that is the preferred description. From regular Euclidean geometric algebra, we know that the versor (or rotor) performing the rotation is simply:

$$R = e^{-\mathbf{B}\phi/2} = \cos(\phi/2) - \mathbf{B}\,\sin(\phi/2)$$

(where we took $\mathbf{B}$ to be a unit bivector, and ϕ the rotation angle in radians) and it rotates an arbitrary object as $X \;\mapsto\; R\,X/R$.

In CGA, we can go further, since we have a way of translating *any* element, even a rotor. A rotation over an axis that goes through a point $\texttt{pt}(\mathbf{t})$ rather than through `no` is simply the translation of the versor:

$$T\,R/T$$

with $T \equiv \texttt{tv}(\mathbf{t})$. But this can be rewritten to:

$$T\,R/T = T\,e^{-\mathbf{B}\phi/2}/T = e^{-(T\,\mathbf{B}/T)\phi/2}$$

and of course the exponent is the dual of the translated origin axis, which is the actual translated rotation axis ℓ':

$$-\texttt{dual}(T\,\mathbf{B}/T) = T\,(-\texttt{dual}(\mathbf{B}))/T = T\,\ell/T = \ell'.$$

So in the end, we just define an axis ℓ' (for instance as the line through two points), multiply its dual by a scalar weight that is proportional to the rotation angle, exponentiate to $\exp(-\texttt{dual}(\ell')/2)$, and we have our rotor. Note that the separation of the points gives a weight to the line, which is used as a rotation angle.

```
x = pt(2*e1), y = pt(2*e1+e3),
A = no^ni,                              // flat point in origin
dynamic{axis = ori(x^y^ni), },
dynamic{V = exp(-dual(axis)pi/5); }
```

Check the dynamic statements to see if they still contain `vtrail(V,A,5),` if not (and only then!) do `dynamic{ vtrail(V,A,5),}`.

For the moment, don't touch the points `x` and `y` (retype them if you already did). Move `A` around and change your viewpoint to convince yourself that this indeed performs a rotation. And by all means, change `A` into another object, for instance the line:

```
A = no^e2^ni,
```

again enjoying the nice property of geometric algebra that the same operator works on any object.

To make different rotations, you can grab the points `x` and `y` on the axis, and move them. Note that their distance is proportional to the rotation angle, (it defines the scalar ϕ in our description above) and their connection to the location and attitude of the rotation axis.

Similarly to translations, there is another way to look at rotations: *a rotation versor is the ratio of two planes*. The rotation is then of course around the intersection of the planes. In order to have some interactive control over those planes, we construct them using a line and a point:

```
cld(); clf();
z = w = pt(e2), label(z); label(w);
line = no^e1^ni,
dynamic{ plane1 = no_shade(z^line), };
dynamic{ plane2 = no_shade(w^line),};
dynamic{ V = plane2/plane1, },
dynamic{ vtrail(V,A,10),};
A = pt(e3),   // or your own choice of object
```

Since `z` and `w` are initially identical, this determines an identity rotation. As you move them apart, you will see that the rotation `R` is over twice the angle between the planes, around their intersection.

6.4 General rigid body motions

A general rigid body motion consists of a rotational part and a translational part. It is completely determined by how one line transforms to another. As you might expect by now, the ratio of two lines is a motion operator that can be used in a sandwiching manner.

```
clearall();
line1 = no^e1^ni, w = no,
dynamic{line2 = no^w^ni,};
dynamic{V = line2/line1,};
dynamic{ vtrail(V,A,10),};
A = no^e1^e3, // or another shape if you prefer
```

Manipulate the lines (`w` is the handle on `line2`, `line1` can be translated freely) to change the rigid body motion. You should see that this generates a screw motion along the perpendicular connection of the lines, with an angle determined by their relative attitudes. The rotation is over *twice* the angle, and the translational pitch over *twice* the shortest vector connecting the lines. Look at the lines 'from above' and move `w` to see that the rigid body motion changes from a screw motion to a translation, and then to an opposite screw.

Now try $\mathtt{A} = \mathtt{e1} \wedge \mathtt{e2} \wedge \mathtt{ni}$ as an object. Can you explain its behavior? There are some pretty patterns that can be made if you choose your axes right. How about making some elements with symmetries of the crystallographic space groups...

But screws can also be specified directly using the exponential notation. If you want to screw around a unit axis through the point $\mathtt{p}$ with attitude $u = \mathbf{u} \wedge \mathtt{ni}$ (with $\mathbf{u}$ a Euclidean unit vector) with pitch τ, this is generated by the screw:

$$\exp(-(\tau\,\mathbf{u} \wedge \mathtt{ni} + \mathtt{dual}(\mathtt{p} \wedge \mathbf{u} \wedge \mathtt{ni}))/2)$$

So we define $\mathtt{p}$ and u and let the norm of u give us τ. To use the screw spinor in various amounts, we give you a slider, and to show more clearly what it does we show the axis:

```
clearall();
u = e1^ni,
p = no,
dynamic{V: V = exp( -phi (u + dual(normalize(p^u)))/2 ),};
dynamic{ axis = ori(p^u),};
dynamic{ vtrail(V,A,25),};
ctrl_range(phi = 0, -1, 1);
A = no^e1^e2,
```

A screw does have 6 degrees of freedom: you see `u` gives us 3, `p` gives us only 2 more (since we can translate along the axis without changing the screw) and the slider gives the sixth.

6.5 Non-Euclidean transformations

In this tutorial, we focus on the Euclidean transformations, plus things like reflections (even spherical inversions) and projections. The conformal model is richer, though, it also contains all *conformal transformations* of space as versors. A conformal transformation is one that preserves angles between transformed objects, and translations, rotations and general rigid body motions do so. But so do scalings relative to a fixed point. The scaling by a factor γ relative to the origin is the versor

$$V = \exp(\mathtt{no} \wedge \mathtt{ni} \log(\gamma)/2)$$

Set up a `vtrail` and try it on the flat point $\mathtt{A} = \mathtt{pt}(\mathtt{e1}) \wedge \mathtt{ni}$.

But some very wild conformal transformations can be made, still as exponentials of bivectors. Try `DEMOloxodrome();`, which demonstrates a 'loxodromic Möbius transformation' (see e.g. [2]), pushing objects from one point to another along a double spiral (drag the circle around). It is a batch running `vtrail`, so you can redefine `A`. After playing a bit, you should kill the dynamic statement keeping it in the plane by `cld(camori)` and play with it in 3D. Or redefine `V` as an exponent of another bivector, and see what you get. Such mappings have been well studied in the complex plane, but to make them with real elements, and in 3D, may well be new.

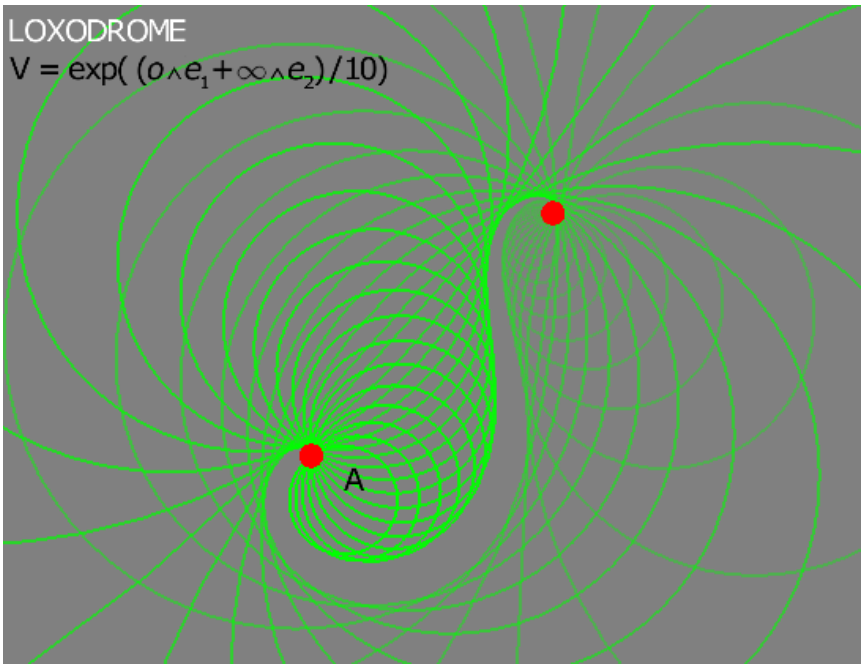


Figure 6: *A loxodromic Möbius transformation on circles.*

6.6 Projections

Let us set up a dynamic statement to experiment with projections.

```
clearall();
dynamic{ XP = (X.P)/P,}
```

To confirm that it works, let us project a flat point onto a plane.

```
X = no^ni,
P = dual(e1),
```

Rotate the camera to see this plane! Drag `X` or `P` to see that this is what you would expect. Change `X` to a line of your choice. Change `P` to a sphere. You can monitor the orientations as well, using `ori()`:

```
cld();clf();
dynamic{ XP = no_weight(ori( (X.P)/P )),}
X = no_weight(ori( no^e1^ni )),
```

Study what happens when you change `P` to its dual and note that it is very essential that you use the direct version of the subspace you want to project to.

You might have expected that you could also project a point (as dual sphere), but setting `X`= `no` shows that that doesn't quite work. A nice surprise is to take a circle, such as:

```
P = dual(e1),
X = ori( (no+ni/2)^(e1+e2)^e3),
```

Drag the circle around. It 'projects' to a circle, not to an ellipse as you might have expected. Of course, in hindsight, it could not have been an ellipse: projection is an outermorphism (see the GABLE+ tutorial), so k-blades get transformed to k-blades. And an ellipse is not a k-blade. But what circle is the result of the projection? Change the parameters of the circle to see what is going on.

To confirm your impression, generate a soft display of the intermediate object $\texttt{dual(P)} \wedge \texttt{X}$ using

```
dynamic{ Xp = alpha( dual(P)^X, 0.2 ),}
```

You see that the projection 'rains down orthogonally' onto the space you project to. Further experiments confirm this. For instance, try $\texttt{X} = \texttt{ori}(\texttt{no} \wedge (\texttt{e1} + \texttt{e2}) \wedge (\texttt{e3}))$. Also, confirm that this still happens for spheres such as $\texttt{P} = \texttt{dual}(\texttt{no} - \texttt{ni}/2)$.

Now study the projection to a circle:

```
P = (no+ni/2)^e1^e2,
X = no^ni,
```

Project a line $\texttt{X} = \text{ori}(\texttt{no} \wedge (\texttt{e1} + \texttt{e3}) \wedge \texttt{ni})$ onto the circle, and find out by dragging when the projection changes orientation. This is tricky! You may help your analysis by visualizing the scalar $\texttt{X} \cdot \texttt{P}$, for instance as the length of a fixed vector by typing $\texttt{dynamic}\{\texttt{showit} = \texttt{X} \cdot \texttt{P} \wedge \texttt{e3} \wedge \texttt{ni}, \}$.

Project a tangent vector onto a circle to finish it all off. What you have seen in this section is that 'flat' blades project onto flat blades precisely as you would hope, but that the Euclidean projection of the other elements is very much sphere-based. The operation is closed within the framework, but we do miss out on the conic sections. We are interested in applications that use these new projections: since they are so unexpected classically, they may not have been seen as tools to simplify certain computations. You can do original research here!

7 Kinematic structures in robotics

Let us build up a kinematic structure: a number of limbs connected by joints which we can control.

We have prepared the structure of a PUMA robot in the file `DEMOpuma.g`, which got loaded in when you loaded in the tutorial directory. It sets up an initialization of a PUMA robot, and then animates this using the global variable `atime` (which keeps track of real time). Basically, its contents is:

```
DEMOpuma_init();         // initialization
dynamic{puma(atime),}    // setting up a function of real time
start_animation();      // animating in real time
```

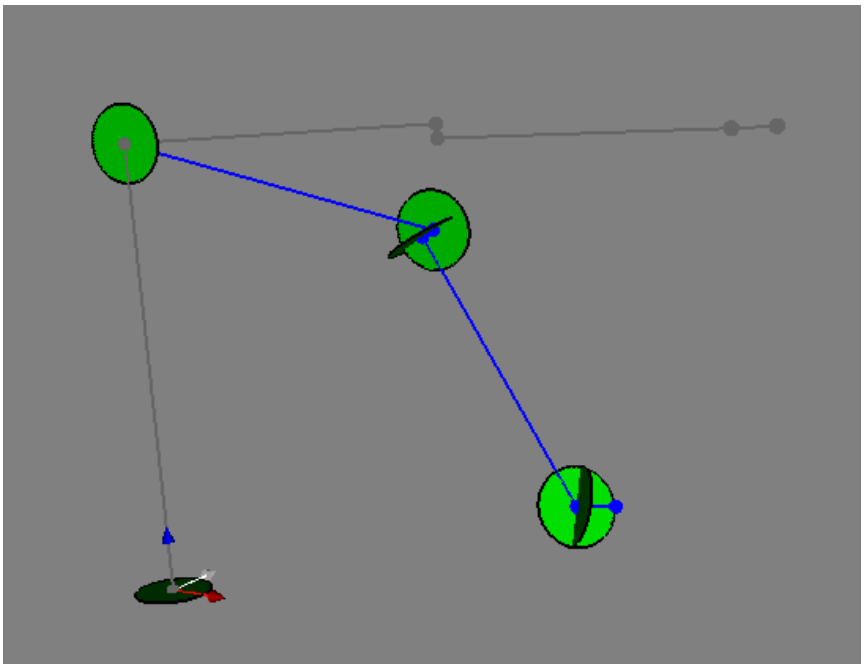


Figure 7: *A puma robot in CGA.*

You can rerun the animation, using either 'Dynamic→Start/resume animation' or by typing `start_animation()`. Note that you can use the scalar controls panel to enable or disable various joints.

If you are interested, open the file `DEMOpuma.g` to see how all elements in this drawing are true objects in CGA (the finite rods we have drawn using one of the drawing modes of point pairs). Here is some detail to help you figure out how it is all done.

> In the function `DEMOpuma_init();`, the `T` and `R` parameters are taken from the commercial specs of the PUMA robot. They denote the translation and orientation of the various joint centers, with the translation always relative to the previous center. The corresponding translation vectors therefore need to be multiplied to provide the total translation; this is the computation for `limb[i]`. These should be used to compute where the rotation axes are, that is the computation for `axis[i]`. Note that these axes are the complete description of the kinematics, they contain all rotation and translation information. These axes are enough to do robot computations, but in order to draw the shape we have picked some points and tangent bivectors; these are `a`[`i`] and `b`[`i`]. The point pairs `a[i-1]` ∧ `a[i]` are what we draw as the robot 'shape' `shape[i]` (by choosing a draw mode that connects them). The initialization ends by specifying which joints participate in the animation by default.
>
> The dynamic part is `puma(atime)`. Some arbitrary but sensibly chosen functions are given which change the joint angles `::t[i]` as a functions of the

real time. Then these are used to compute the actual rigid body motion versors `M[i]`, from the base upwards. As you see, only `axis[i]` is required. To draw the robot, we use these `M[i]` to compute the drawing items: the rods and the bivectors. They of course transform in exactly the same way.

This is all very much like the classical way of treating a robot, but there is a difference in the generality of the transformations. Any element of CGA can be transformed using the versor description, so the drawing of the tangent bivectors for the joints is exactly the same transformation as for the connected point pairs denoting the limbs. If there were any other items we would like to draw (such as a mesh indicating the arm shape), they would obey the same transformations. That is the major difference with the classical matrix method to process the same results: the matrices would only work on position vectors, all other elements would have to be transformed using their own transformations (often derived from these matrices, but still cumbersome), or first reduced to point concepts.

You should learn to take advantage of these new possibilities of CGA, by daring to define your shapes as more than simply a mesh of points.

8 Summary and conclusion

This concludes our cursory exploration of the conformal model of Euclidean geometry. As you have seen, many familiar but not-so-well defined elements from practical geometry find a home here, and are enriched by their relationship with other elements. Operations are much more powerful and general in their versor representation. There are also some things we haven't seen before: strange 'Euclidean projections' and some non-Euclidean transformations.

We have not touched upon the corresponding calculus at all, and have done very little that could be seen as an 'application'. Our main goal was to give you a good understanding of the basics: why it works, and what makes it different from more classical methods.

We expect to augment the tutorial over the years to explain new views, discoveries, or techniques in working with CGA. So, keep watching `http://www.science.uva.nl/ga/`...

References

[1] L. Dorst, *Classification and parametrization of blades in the conformal model of Euclidean geometry*, in preparation, will appear on `www.science.uva.nl/ga/`.

[2] T. Needham, *Visual Complex Analysis*, Clarendon Press, Oxford, 1997.

[3] GABLE + tutorial at `www.science.uva.nl/ga/tutorials`

[4] S. Mann and L. Dorst, *Geometric algebra: a computational framework for geometrical applications (part II: applications)*, IEEE Computer Graphics and Applications, vol.22 no.4, July/August 2002, pp.58-67.

A The homogeneous model

In the classical use of linear algebra, you have probably come across *homogeneous coordinates*. It is trick to linearize useful transformations occurring in robotics and graphics, such as translations and projections. It works by embedding the 3D Euclidean space into a space of 1 more dimension. A point $\mathbf{p}$ with coordinates $(p_1, p_2, p_3)^T$ thus becomes represented by the vector $(p_1, p_2, p_3, 1)^T$. Now a translation is a linear operation, for the representation of $\mathbf{p} + \mathbf{t}$ can be achieved by a matrix multiplication.

In geometric algebra, you would describe the embedding not in terms of coordinates, but by addition of an extra vector, say $\mathbf{e}_0$, which is orthogonal to all Euclidean dimensions (so $\mathbf{e}_0 \cdot \mathbf{e}_1 = 0$ et cetera). It is not clear how you should choose the norm of $\mathbf{e}_0$, but for invertibility we may demand $\mathbf{e}_0 \cdot \mathbf{e}_0 = 1$.

Then the rest of the machinery of geometric algebra can generate quite a bit more than merely the representation of points. Indeed, the outer product $p \wedge q$ of two point representatives p and q has all relevant elements to represent the directed line through p and q, as is easily seen:

$$p \wedge q = (\mathbf{e}_0 + \mathbf{p}) \wedge (\mathbf{e}_0 + \mathbf{q}) = \mathbf{e}_0 \wedge (\mathbf{q} - \mathbf{p}) + \mathbf{p} \wedge \mathbf{q}$$

Here we recognize the direction vector $\mathbf{q} - \mathbf{p}$ and the moment $\mathbf{p} \wedge \mathbf{q}$, from which one can compute the support vector $(\mathbf{p} \wedge \mathbf{q})/(\mathbf{q} - \mathbf{p})$ connecting the line perpendicularly to the origin and therefore specifying the location. In fact, the 6 coordinates of $p \wedge q$ are the *Plücker coordinates* of the line, known in advanced computer graphics – but as we see, actually just as elementary as homogeneous coordinates.

More details may be found in our written tutorial [4]. For now, you have enough to understand `DEMOhomogeneous();`, which visualizes the above for a 2D geometry (so that the visualization is 3D).

The bottom line is that the use of homogeneous coordinates allows us to represent translated subspaces in n-D by subspaces through the origin in $(n+1)$-D (mathematicians call those 'homogeneous subspaces' and that explains the name of the model).

If you want to play more with this particular geometric algebra, you get it by setting

```
default_model(p3ga)
```

(for 'projective 3D geometric algebra). The unit vector for the extra dimension is `e0`. A demonstration of its projective use may be found in appendix B. There you also find that this model is really contained within the conformal model: it is the algebra of all flat elements (i.e. elements containing a factor `ni`). To show this, consider the representation of flat points in the conformal model:

$$p \wedge \mathtt{ni} = (\mathtt{no} + \mathbf{p} + \tfrac{1}{2}\mathbf{p}^2\mathtt{ni}) \wedge \mathtt{ni} = (\mathtt{no} + \mathbf{p}) \wedge \mathtt{ni} = ((\mathtt{no} - \mathtt{ni}/2) + \mathbf{p}) \wedge \mathtt{ni} = (\mathbf{e}_0 + \mathbf{p}) \wedge \mathtt{ni}$$

so that when we define $\mathbf{e}_0 = \mathtt{no} - \mathtt{ni}/2$ (which gives $\mathbf{e}_0^2 = 1$), the homogeneous model is indeed contained within the algebra of all elements with a factor '$\wedge$`ni`'.

B Embedding projective geometry into CGA

We will show how the projective operations are embedded in CGA. We do it twice, in two different models of geometry, which in the viewer are called `p3ga` and `c3ga`.

B.1 P3GA projection

Projective operations are quite naturally embedded in the 3D projective geometric algebra, i.e. a 4D model of 3D geometry with an extra homogeneous dimension. Let us study that by going over to the `p3ga` model, which has `e0` as the basis vector in the extra dimension providing the homogeneous coordinate. It has multiplication table

	e0	e1	e2	e3
e0	1	0	0	0
e1	0	1	0	0
e2	0	0	1	0
e3	0	0	0	1

You don't need to type the following commands, they are in the file `DEMOp3ga.g` which you loaded in with the rest of the tutorial files. Simply type

```
DEMOp3ga();
```

But to understand the principles we try to elucidate in this chapter, we need to look in more detail at the contents of `DEMOp3ga.g`. Here is a stripped version of the essence.

```
// simplified contents of DEMOp3ga.g
// simple demonstration of projection in the p3ga model

function proj(x,O,I) { return no_weight(dual(I).(O^x));}

batch DEMOp3ga()
{
        cld();clf();clc();
default_model(p3ga);
O = e0,  label(O); // pinhole camera
I = no_shade(alpha(p3ga_point(e1)^e2^e3,0.5)),    // image plane

x = p3ga_point(2*e1+e2), label(x);
y = p3ga_point(2*e1+e3), label(y);
dynamic{ line  = x^y,}
dynamic{ xp    = proj(x,O,I),}
dynamic{ yp    = proj(y,O,I),}
dynamic{ xr    = alpha(O^x,0.3),}
dynamic{ yr    = alpha(O^y,0.3),}
```

```
dynamic{ linep = proj(line,O,I),}
}
```

The objects that are formed are non-metric: distances after projection have no linear relationship to those before projection. It may seem that the projection function uses the metric in the inner product, but that is merely an efficient way of computing the *meet*: we could also have written $-\texttt{dual}(\texttt{dual}(\texttt{I}) \wedge \texttt{dual}(\texttt{O} \wedge \texttt{x}))$ as a perhaps more obviously non-metric implementation.

B.2 C3GA projection

We would like to do projection in CGA, because in many applications such as stereo vision we need to combine projectively obtained images to get an optimal interpretation in some Euclidean sense. The former is PGA-like, the latter resides naturally in CGA.

So how do we embed a construction like the above in CGA? Somehow, we need to reconcile the metric aspects of CGA with the non-metric aspects of PGA; we need to 'forget' the metric for the projective operations, and restore it afterwards (for we are allowed to talk about metric aspects in the image plane by itself). There is a mathematical clue in the fact that PGA is a subalgebra of CGA: if you have an element X of PGA, then it behaves very much like the element $\texttt{ni} \wedge X$ of CGA. (PGA-points are like flat points in CGA, a PGA line is the outer product of two points but a line in CGA has an extra '$\texttt{ni}\wedge$' et cetera.) Conversely, 'stripping off' the `ni` from a CGA element gives a PGA-like element. And clearly, not all elements can be so embedded: rounds do not have a `ni` factor – and indeed, they are truly Euclidean elements so we would not expect PGA capable of treating them.

But this is a slightly tricky embedding, for it is not an outermorphism: the outer product of two embedded points is *not* the embedding of their outer product. In fact, the outer product in CGA of two elements of the form $\texttt{ni} \wedge X$ is obviously always zero. When 'doing something projective' you therefore consciously have to move to the PGA – if you don't, you get a rather different kind of 'projection', which we saw in section 6.6.

How do we 'strip off `ni`' from a CGA element, and arrive at PGA? We would like a PGA in which the element `e0` (representing the point at the origin) has enough of an inverse to allow us to do dualization in a standard manner; for instance by demanding $\texttt{e0} \cdot \texttt{e0} = 1$, as is done in `p3ga`. Of course `e0` should also be orthogonal to all Euclidean elements. In brief, `e0` should be set to:

$$\texttt{e0} = \texttt{no} - \texttt{ni}/2$$

After some algebraic simplification we get the following casting function, which again you don't need to type since it is in the file `DEMOprojective.g`, which you loaded in with the rest of the tutorial files.

```
// part of DEMOprojective.g
// from CGA to 'PGA-like within CGA'
function c2p(A) {
```

```
  if (ni^A == 0)
    return ( no.A + ni^(ni.(no.A))/2 );
  else
    return cprint("This metric element cannot be made projective!");
}

// from 'PGA-like within CGA' to CGA
function p2c(A)  { return ni^A; }

// actual projection
function pro(X,O,I)
  { return no_weight(dual(I).pouter(O,X) pow(-1,grade(X)));};
```

The non-linear aspect is the `else` part, which returns 0 for any element not of the form `ni` $\wedge$ `A`. This is where we refuse to project essentially metric elements such as rounds (including points) and tangents. Now you can make the connection line between two flat points `x` and `y` as:

```
x = c3ga_point(e1)^ni, y = c3ga_point(e2)^ni,
line = p2c( c2p(x) ^ c2p(y) ),
```

You could draw the elements before casting them using `p2c` (since they are well-defined elements of CGA), but it is very confusing so we would strongly advice against it. (Oh, go ahead – try it if you must. But don't say we did not warn you.)

The first example, now fully done in CGA, becomes like this in `DEMOprojective();`:

```
// other part of DEMOprojective.g
function pouter(X,Y) { return p2c(c2p(X)^c2p(Y));}, // projective outer product

function proj(X,O,I) { return no_weight( dual(I).pouter(O,X) pow(-1,grade(X)) );},

batch DEMOprojective()
{
function p2c,c2p,pro;

cld();clf();clc();
default_model(c3ga);
O = black(no^ni), label(O);         // flat point for the pinhole
I = color(pt(e1)^e2^e3^ni,1,1,0.6,0.5), // flat image plane

x = pt(2*e1+e2)^ni, label(x); // now we must use flat points
y = pt(2*e1+e3)^ni, label(y);
dynamic{ line  = pouter(x,y),},
dynamic{ xp    = proj(x,O,I),},
dynamic{ yp    = proj(y,O,I),},
dynamic{ xr    = alpha(pouter(O,x),0.3),},
```

```
dynamic{ yr    = alpha(pouter(O,y),0.3),},
dynamic{ linep = pro(line,O,I),},
}
```

Note that the colors have changed relative to the `p3ga` implementation, since all elements are of 1 grade higher in `c3ga` due to the '`ni`$\wedge$' factor.

We defined the `pouter` function to show that the basic structure is as before. The crucial function here is of course the projection function, which we made here as a meet in CGA between a (flat) line and the (flat) image plane `I`.

> We could also have set up the meet in PGA, but this is more involved, because `dual()` is relative to the pseudoscalar of CGA (which is the default model). So we would first have to construct a pseudoscalar for our PGA model and use that for the dualization. This gives the functions:
>
> ```
> function pdual(X) { return dual( c2p(X),c2p(no^e1^e2^e3^ni) ); }
> function pro(X,O,I) {
>    return  no_weight( p2c( pdual(I).(c2p(O)^c2p(X)) ));
> }
> ```
>
> Note that the inner product does not need to be corrected to some new '`pinner`' after our careful mappings `c2p` and `p2c`.

As we stated in section 6.6, you do no get conic sections as elementary objects in CGA, so these familiar parts of projective geometry do not get a novel treatment in CGA.