A Modern Matlab Tool For Teaching Projective Geometric Algebra

Zachary Leger[1][0009−0003−8831−6834] and Stephen Mann[2][0000−0001−8528−2921]

[1] University of Waterloo, Waterloo ON N2T 1S1 Canada
zcjleger@uwaterloo.ca
[2] University of Waterloo, Waterloo ON N2T 1S1 Canada
smann@uwaterloo.ca
https://cs.uwaterloo.ca/~smann/

Abstract. We detail the creation of PGABLE, a MATLAB package to aid in the teaching of Projective Geometric Algebra (PGA). We begin by discussing the precursor to PGABLE created in 1999, GABLE, and motivating the creation of PGABLE. Next, we discuss the various internal aspects of GABLE that needed to be modified to create PGABLE, as well as to update the software to use features of the more modern MATLAB. We then discuss drawing graphical representations of the PGA objects, and in particular how we draw objects at infinity. We then briefly describe the PGABLE tutorial we created to teach PGA.

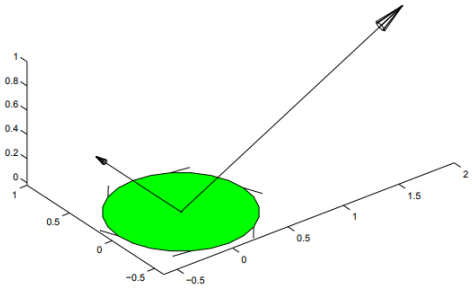
Keywords: PGA · Geometric Algebra · MATLAB.

1 Introduction

GABLE was a prototype MATLAB package supporting a few geometric algebras written in 1999. The main algebra used in GABLE was $\mathbb{R}^{3,0,0}$, Ordinary Geometric Algebra (OGA), although some support was available for the homogeneous model $\mathbb{R}^{2,1,0}$. In addition to being a prototype geometric algebra package, GABLE was designed to support a tutorial introduction to geometric algebra [1]. However, being a prototype, little work has been done on GABLE since 1999.

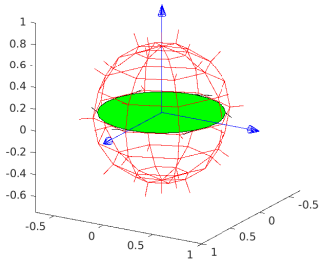
GABLE used MATLAB's command line interface to give users a rich mathematical setting in which to explore geometric algebra. In addition to writing equations, GABLE used MATLAB graphics to draw various objects, including vectors, bivectors, and trivectors (Figure 1).

In recent years, Projective Geometric Algebra (PGA) has become a popular model in the geometric algebra community to represent 3D rigid body motions [2]. As PGA is a 4D geometric algebra (the space $\mathbb{R}^{3,0,1}$), GABLE cannot support it. The goal of this work is to update and extend GABLE into PGABLE, a modern MATLAB package able to support PGA. In particular, the graphical capabilities of MATLAB have improved since 1999, and the support for object



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

(a)



(b)

Fig. 1: GABLE. (a) Graphics output of example text input (below) ; (b) Drawing of three vectors (blue), a bivector (green) and a trivector (red).

oriented programming in MATLAB has changed signficiantly since then. In addition, we have created a tutorial alongside this software package to introduce users to PGA.

In this paper, we discuss various features of PGABLE. While we give some detail on PGABLE MATLAB commands, the goal of this paper is to discuss the main ideas and challenges in implementing PGABLE and is not a comprehensive guide to PGABLE. Further, while PGABLE has support for both OGA and PGA, this paper will focus on the PGA apects of PGABLE. Since we continue to develope PGABLE, note that this paper is based on PGABLE version 1.1.0.

2 GABLE Implementation Overview

In GABLE, a multivector A is stored as the 8×1 column matrix that contain the coefficients of the basis blades

$$1, \;\; e_1, \;\; e_2, \;\; e_3, \;\; e_1 \wedge e_2, \;\; e_2 \wedge e_3, \;\; e_3 \wedge e_1, \;\; e_1 \wedge e_2 \wedge e_3 \tag{1}$$

For example, the multivector $e_1 + 2e_2 \wedge e3$, would be stored as $\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 2 & 0 & 0 \end{bmatrix}^T$. The basis vectors e_1, e_2, e_3 were implemented as functions in MATLAB, with each function returning a GABLE element storing the corresponding 8×1 matrix for that basis vector. We denote by $A.m = [A_0, A_1, A_2, A_3, A_{12}, A_{23}, A_{31}, A_{123}]^T$.

The geometric product $A\,B$ of two multivectors A and B is computed by converting A into an 8×8 matrix, which will denote $[A]$, that is the linear function computing "the geometric product $A*$". The geometric product operation creates and returns a GA element storing the resulting column vector from the matrix product $[A]B.m$. The inner product and the outer product are implemented in a similar manner. MATLAB objects were used to overload the arithmetic operators `*` (geometric product), `^`(outer product), `+` (addition). The inner product was implemented as a MATLAB function `inner()`.

Elements of the algebra are drawn by explicitly calling a `draw()` routine. Lines on bivectors and on trivectors are used to indicate the orientation (sign) of the object. GABLE provided a variety of routines for rotating objects, although with more modern versions of MATLAB, rotation and scaling of objects in the graphics window can be done using the mouse.

3 Extending GABLE to PGABLE

Many of the design decisions used in GABLE were kept in PGABLE: multivectors are represented with column matrices; products of two multivectors are computed by expanding the column of one element into a square matrix and multiplying the column matrix of the other element by this square matrix; and the graphical representation of basic elements (vectors, bivectors, and trivectors) remains the same. MATLAB has evolved since 1999, and a variety of changes were made to update GABLE to current MATLAB, mostly dealing with the organization of objects and methods. In particular, classes, which were implicitly represented in 1999, are now explicitly represented in the MATLAB language, and static methods were introduced.

We note that SUGAR [9] is another MATLAB package for Geometric Algebra that provides support for PGA, that was developed concurrently with PGABLE. SUGAR provides support for more algebras than PGA and OGA, and has some graphics support for the Conformal Model, but none (at the time of this writing) for PGA. Regardless, we chose to develop PGABLE to focus on PGA and a tutorial for learning PGA, and to provide graphics support for visualizing PGA objects. Thus, writing a package that created only these two algebras seemed a better choice for our tutorial.

3.1 Dimensionality Increase

With the addition of e_0 in PGA, elements in PGA must be represented by an 16×1 column matrix. We chose to use the basis

$$1,\ e_0,\ e_1,\ e_2,\ e_3,\ e_0 \wedge e_1,\ e_0 \wedge e_2,\ e_0 \wedge e_3,\ e_1 \wedge e_2,\ e_1 \wedge e_3,\ e_2 \wedge e_3,$$
$$e_0 \wedge e_1 \wedge e_2,\ e_0 \wedge e_1 \wedge e_3,\ e_0 \wedge e_2 \wedge e_3,\ e_1 \wedge e_2 \wedge e_3,\ e_0 \wedge e_1 \wedge e_2 \wedge e_3$$

With this change in data structure, the pre-existing operations such as the geometric product, outer product, inner product, inverse and reverse needed to be rewritten. However, the underlying mathematics remained the same. It is simply the case that the 8×8 matrices were replaced by mathematically analogous 16×16 matrices.

In PGABLE, `e0`, `e1`, `e2` and `e3` are used for e_0, e_1, e_2 and e_3 respectively. Larger degree blades can be specified using the outer product of these vectors, such as `e0^e1`; however, these basis blades also have a concatenated shorthand: one can use `e01` in lieu of `e0^e1`. Additionally, `I3` and `I4` can be used for the pseudoscalars of OGA and PGA respectively.

3.2 Rendering Redesign

Traditional visual representations of geometric algebras draws scalars as points, vectors as lines, bivectors as planes, and trivectors as volumes, all of which are centered at the origin. PGA is a model of geometric algebra which can represent points, lines, planes and volumes offset from the origin. This means the rendering abilities of GABLE had to be extended to draw elements with this potential offset.

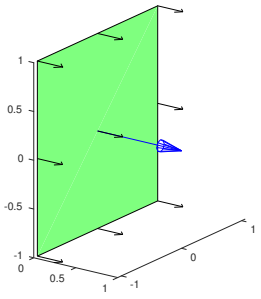


Fig. 2: In OGA, e_1 represents the blue vector; in PGA, e_1 represents the green plane.

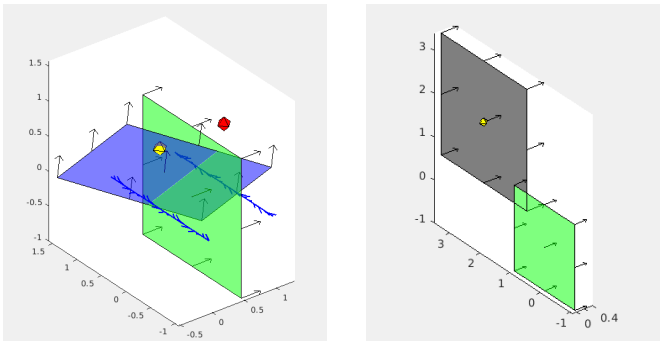
Extending the rendering abilities alone did not suffice to accurately represent PGA. PGA is fundamentally different than most other geometric algebra models (such as the homogeneous model or the conformal model) in that vectors represent planes, not lines or points. Additionally, bivectors represent lines and trivectors represent points in PGA. This means that some elements possess a fundamental ambiguity as to how to represent them, depending on whether we are considering the OGA interpretation or the PGA interpretation. For example, consider the element e_1. In the traditional view, this is represented by a unit arrow pointing along the positive direction on the e_1 axis. However, in the PGA view, this element represents the (oriented) plane whose *normal* is e_1 (Figure 2). For this reason, each element in PGABLE is associated with its model (OGA or PGA).

The user is also associated with a model, which is the model in which they intend to work. When the user creates an element, that element is associated with the user's current working model (unless they explicitly state otherwise). Hence, if the user creates an element, say e_1, that element will be associated with OGA if they are in OGA mode, and with PGA if they are in PGA mode. Once the element is created, calling `draw` on the element will draw the geometric interpretation of the model associated with it (regardless of the current state the user is in).

PGABLE objects are drawn using the `draw` command. Changing the color and offset of elements can be done via arguments in the draw command. For

example, `draw(e1,'r')` draws the vector e_1 in red. We also allow the user the more advanced option of being able to pass MATLAB optional drawing arguments into the draw command for more customization abilities.

In PGABLE, we draw a plane as a semi-transparent square with arrows indicating the direction of the plane normal; we draw lines as line segments with "hairs" to indicate the orientation of the line; and we draw points as small octahedrons. Figure 3 (a) shows an example of these objects drawn in MATLAB. We draw the planes as semi-transparent to allow the user to see objects on the other side of the planes. We draw the hairs on the lines rotating around the line so that the line direction is visible from any viewing angle. Furthermore, we decided to draw the points as a polyhedron to allow points to be more visible when they lie on a plane; when we draw the points with any of the various MATLAB point markers, it became hard to tell the position of a point relative to a plane. We chose an octahedron to keep the polygon count low and maintain fast rendering speed.



(a) (b)

Fig. 3: Drawing in PGABLE (a) Two points, two lines, and two planes; (b) the default square drawn for a plane (green) and the same plane scaled by a factor of 2 (black) centered at the yellow point.

By default, we set the the square for a plane to be 4 square units, centered at the point on the plane closest to the origin. However, one can draw a larger square for a plane by changing the scale factor of the plane; e.g., `draw(4*e1)` will draw the square to have four times the area of the square drawn for `draw(e1)`, and one can give an optional point C as an argument when drawing a plane n, and the center of the square drawn for n will be the closest point on n to C. Figure 3 (b) shows the default square drawn with `draw(e1)` and an offset square drawn with `draw(2*e1,gapoint(0,2,2),'k')`. Similarly, the line segment drawn for a line is scaled by the norm of the line, and the line segment may also be drawn relative to an arbitrary point.

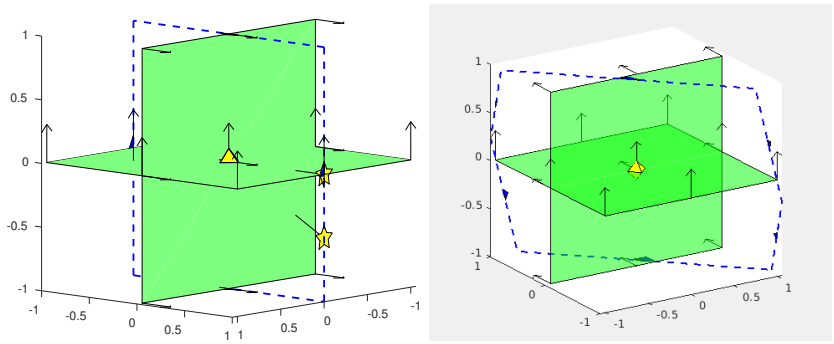
3.3 Degenerate Objects

PGA has lines and points at infinity. Rendering such objects in MATLAB is a challenge. In particular, MATLAB computes a bounding box for all objects to be drawn, and creates an axis aligned box in which to draw these objects. Thus, drawing objects outside this box (using MATLAB's standard drawing routines) would result in MATLAB creating a bigger bounding box. A second issue is choosing where these objects at infinity should be drawn. Technically, where they appear depends on the location of the viewer. If we assumed a position of the viewer relative to the screen, points at infinity that are behind the viewer would not appear.

To address these two issues, we designed PGABLE to keep a list of object elements within the scene, and we draw objects at infinity as if the viewer was located at the center of the MATLAB bounding box. Elements that are persistently redrawn based on the size of the bounding box of the figure are referred to as "dynamic items". PGABLE keeps a list of these dynamic items. The list is stored as a persistent variable in a function called `manage_dynamic_items`. As the name suggests, the function `manage_dynamic_items` manages the list of dynamic items. Hence, `manage_dynamic_items` is able to add and remove items from the list, as well as clear and print the list. A MATLAB event listener is employed to listen for changes in the bounding box of the GA Figure (which can be caused by panning and zooming from the user). The event calls the function `manage_dynamic_items` to redraw the dynamic elements of the scene. A similar call is also made whenever any "still" element (non-dynamic element) is drawn to the scene. This special case must be handled separately because the bounding box may change after a new element is drawn, but this change to the bounding box will not trigger the event listener.

Figure 4 (a) shows an example of two planes, a point, a line at infinity, and two points at infinity. A line at infinity is drawn as a dashed line around the boundary of the MATLAB box; points at infinity (which are essentially directions) are drawn as stars on the boundary of the box, with a short line segment to better indicate the direction of the point at infinity.

Drawing objects in this manner in MATLAB raises a variety of issues. First, while lines at infinity often have four line segments around the boundary of the box (Figure 4 (a)), potentially they have as many as six segments (Figure 4 (b)), although the vanishing line will never have fewer than four segments, since this graphical representation of a vanishing line is the intersection of the bounding volume with a plane through the center of the bounding volume. A second issue is that because PGABLE keeps its own list of objects at infinity, the MATLAB `clf` command (which is used to clear the graphics window) does not clear our list of objects at infinity. Thus we had to implement our own `pclf` command to clear these objects at infinity. Note also that drawing the objects at infinity as if the viewer is at the center of the MATLAB bounding box means that if the user draws other objects (not at infinity) that changes the bounding box, then the objects at infinity appear to "move", since the center of the bounding box has changed.



(a) (b)

Fig. 4: Objects at infinity. (a) A line and two points at infinity. (b) A line at infinity drawn with six segments.

3.4 Additional Operations and Details

To show the full capabilities of PGA, we needed to add new computational abilities to GABLE. First, since PGA does not possess the traditional geometric algebra dual (since its pseudoscalar is not invertible), users typically want other types of dualization. We provided the Hodge dual [2] and the Poincaré dual [8].

As the `join` operation is computed differently in PGA than in the standard geometric algebra, we provide a new operation `PGAjoin` that computes

$$\star^{-1}(\star A \wedge \star B),$$

which is the PGA join operation described in PGA4CS [2]. Since the PGA `meet` operation is simply the outer product, for the sake of consistency, we have provided a `PGAmeet` operation that performs a call to the outer product function.

Since PGABLE contains an OGA and PGA mode, we needed a way to convert between PGA elements and OGA elements for each mode. By default, elements of OGA are interpreted as their direct PGA equivalent in PGA mode (for example, the element e_1 in OGA is simply interpreted as e_1 in PGA). This conversion can be accomplished by passing the OGA element through the function `PGAcast`. However, users may want to have their OGA elements converted into PGA elements through a geometric interpretation (for example, the arrow representing element e_1 in GA would be $e_2 \wedge e_3$ in PGA). Thus, we have provided users with the function `geoPGA` to cast OGA elements into PGA using a geometric interpretation.

There are two norms in PGA, the Euclidean norm $||\cdot||$ and the vanishing norm $||\cdot||_\infty$. The `norm` function of PGA computes the Euclidean norm. For the vanishing norm, we provide the function `vnorm`, which is computed by using the equivalence $||A||_\infty = ||\star A||$ as described in PGA4CS [2].

As a side note, epsilon terms frequently occur in numeric computations. Some packages, such as GABLE, omit these terms rather than print them. E.g., rather

than output `e1 + 1e-16*e2`, GABLE prints `e1`. Since at times these epsilon terms cause issues (such as when testing the grade of an object), most packages provide a mechanism to "zero-out" the esplison terms (e.g., `GAZ` in GABLE; `clean` in SUGAR). PGABLE also provides a mechanism for clearing the epsilon terms, but PGABLE prints the epsilon values as ε rather than a numeric value; e.g, rather than print `e1 + 1e-16*e2`, PGABLE prints `e1 + ε*e2`. In addition to shortening the output, displaying these terms as ε has the advantage that the user will know which terms will disappear when the "zero-out" routine is called (e.g., `e1 + ε*e2 + 1e-15*e3` will have its `e2` term zero'ed out, but its `e3` term will remain).

4 Examples

We give two examples to demonstrate PGABLE. The first example is to show how to construct various entities in PGABLE, while the second example shows how straightforward it is to use PGABLE (and PGA) to construct Bézier curves.

As a first example, we start by drawing three planes and intersect these planes to get a point, which we also draw:

```
n1=e1 - 0.5*e0; draw(n1,'r');
n2=e2 - 0.5*e0; draw(n2,'m');
n3=e3 - 0.5*e0; draw(n3,'b');
P = n1^n2^n3;   draw(P);
```

See Figure 5(a).

Next, we directly construct a second point with `gapoint`, and construct a line using the `join` operator:

```
Q = gapoint(0.5,-1,1); draw(Q);
L = join(P,Q); draw(L,'y');
```

See Figure 5(b).

We now build and draw a fourth plane, parallel to one of the first three planes, and intersect the two parallel planes to get a line at infinity:

```
n4=e1 + 0.5*e0; draw(n4,'r');
Li = n1^n4; draw(Li,'g');
```

See Figure 5(c).

We conclude the first example by intersecting our line at infinity with a plane, giving a point at infinity:

```
Pi = Li^n3; draw(Pi);
```

See Figure 5(d).

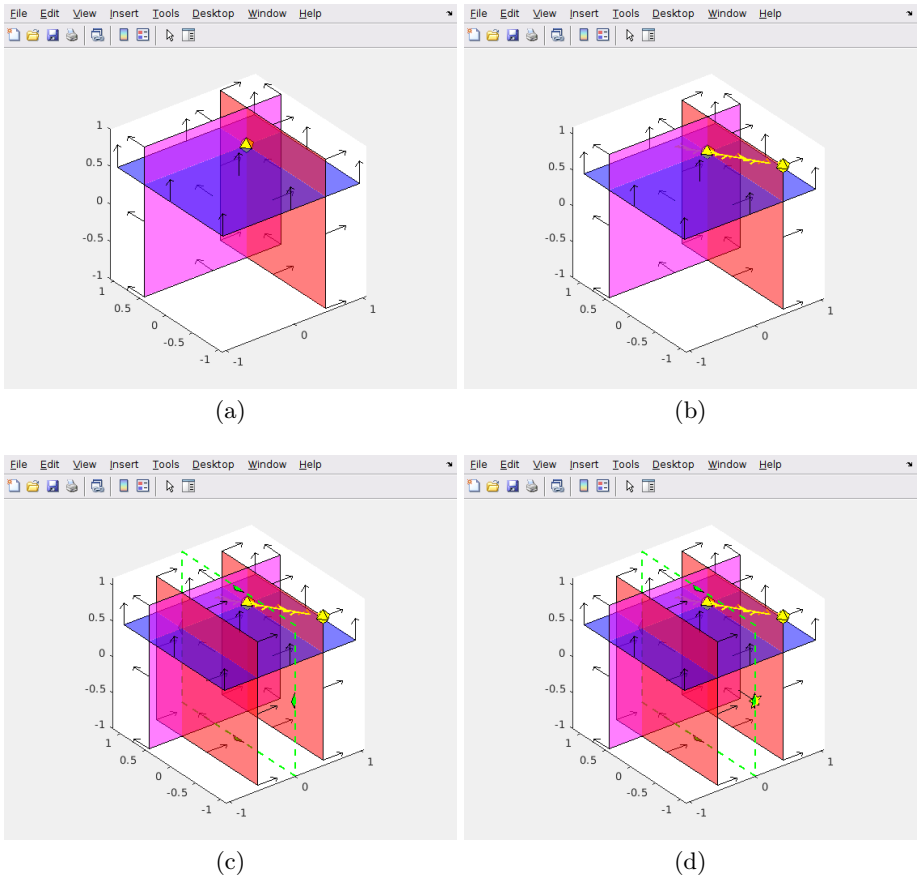


Fig. 5: Figures for PGABLE example, (a) showing planes and a point; (b) constructing a line with `join`; (c) constructing a line at infinity; and (d) constructing a point at infinity.

4.1 Bézier Curves

As a second example, we use PGABLE to evaluate a Bézier curve. A degree n Bézier curve [4] is a polynomial curve with *control points* P_i blended with the Bernstein polynomials $B_i^n(t) = \binom{n}{i}(1-t)^{n-1}t^i$:

$$B(t) = \sum_{i=0}^{n} P_i B_i^n(t).$$

A Bézier curve is typically evaluated at parameter value t using repeated affine combinations with de Casteljau's algorithm [4]:

$$P_i^r = (1-t)P_i^{r-1} + tP_{i+1}^{r-1} \qquad \begin{cases} r = 1, ..., n \\ i = 0, ..., n-r \end{cases} \tag{2}$$

with $P_i^0 = P_i.$, where P_0^n is the point on the curve $B(t)$. Figure 6 illustrates de Casteljau's algorithm. Figure 7(b) gives PGABLE code for evaluating a Bézier curve with de Casteljau's algorithm. The function `bezier` would be called like

```
CP=[gapoint(0,0,0),gapoint(1,0,0),gapoint(1,0,1),gapoint(1,1,1)];
bezier(CP, 0.5)
```

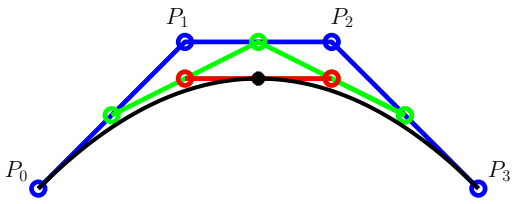


Fig. 6: de Casteljau's algorithm evaluating a Bézier curve $B(t)$ at $t = 0.5$. Each green point is computed as the midpoint of two blue points; each red point is computed as the midpoint of two green points; the black point is the midpoint of the two red points and is $B(0.5)$. The curve itself is shown in black.

Figure 7(b) shows a plot of a Bézier curve in PGABLE; the yellow octahedrons are the control points; the curved line is the Bézier curve, constructed by repeatedly calling `bezier(CP,t)` for `t` varying from 0 to 1. The primary thing to note in this example is that de Casteljau's algorithm (and many other splines contructions) work well in PGA (and PGABLE), with a straightforward mapping from (2) to the code in Figure 7 (a), since affine combinations of points are well defined in PGA.

We have implemented Bézier curves, B-spline curves, tensor product surfaces, and triangular Bézier patches in PGABLE, all of which are available as supplementary material for PGABLE [7]. PGA works especially well with a

```
function r = bezier(cp,t)
  [row,col]=size(cp);
  for i=1:col-1
    for j=1:col-i
      cp(j) = (1-t)*cp(j) + t*cp(j+1);
    end
  end
  r = cp(1);
```

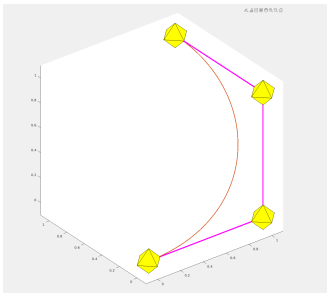


Fig. 7: (a) PGABLE code for de Casteljau's algorithm for evaluating a Bézier curve; (b) A PGABLE plot of a Bézier curve.

de Casteljau evaluation of triangular Bézier patches, since an intermediate step of the surface evaluation algorithm produces three points in the tangent plane of the surface; these points may be `join`ed to produce the tangent plane (Figure 8).

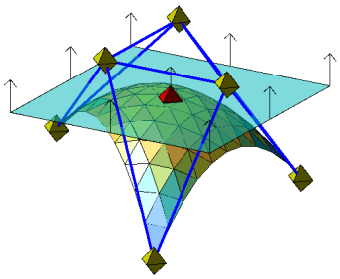


Fig. 8: Triangular Bézier patch with its control points, control net, evaluated at a point in the center of the patch, shown along with the tangent plane to the surface at that point.

5 Discussion and Conclusions

The purpose of writing PGABLE was to provide software for a tutorial introduction to Geometric Algebra, and in particular, to Ordinary Geometric Algebra and Projective Geometric Algebra. The tutorial for OGA was an updated and abbreviated version of the tutorial for GABLE; in particular, we removed the material on the homogeneous model, since that material is supplanted by the material in the section on PGA.

Initially, we wrote the PGABLE code to be as coordinate-free and PGA based as possible. Unfortunately, this led to a variety of issues, such as epsilon

errors and debugging issues since the code was self-referential. Thus, many parts were rewritten to be coordinate based.

The section of the tutorial on PGA focuses heavily on the objects of PGA (planes, lines, and points): how to create them and their visual representation in PGABLE. The tutorial also discusses various operations in PGA (such as intersecting planes; extracting the direction of a line as well as the closest point on a line to the origin), and discusses degenerate objects as covered in Section 3.3. Duality, meet, and join are discussed, and a large section is devoted to transformations. All of these topics are presented through a mix of PGABLE code (for the reader to type and/or to run), mathematical discussion/derivations, and exercises.

In the future, we are planning to change the rendering of lines and planes so that rather than being drawn as an arbitrary line segment and square, we instead draw the portion of the line and plane that lies within the MATLAB bounding box.

Fig. 9: PGABLE being used by students at Howest University

PGABLE is currently being used at Howest University of Applied Science & Arts, Dept Digital Arts and Entertainment, for two courses consisting of 150 students (Figure 9).

PGABLE and the tutorial may be downloaded at

`https://cs.uwaterloo.ca/~smann/PGABLE/`

PGABLE may also be installed as a MATLAB package.

Acknowledgments. This study was funded by NSERC, the Natural Sciences and Engineering Research Council of Canada.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Dorst, L., Mann, S. and Bouma, T. GABLE: A Matlab tutorial for geometric algebra. Available at `https://cs.uwaterloo.ca/~smann/GABLE/`, 1999.
2. A Guided Tour to the Plane-Based Geometric Algebra PGA. Dorst, L. and De Keninck, S. `https://bivector.net/PGA4CS.html`
3. Dorst, L., Fontijne, D., and Mann, S.. *Geometric Algebra for Computer Science*. Morgan-Kaufmann, 2007.
4. Farin, G., *Curves and surfaces for CAGD: a practical guide*, Morgan-Kaufmann, 2001. Fifth edition.
5. Leger, Z. and Mann, S. A Tutorial for Plane-based Geometric Algebra. Available at `https://cs.uwaterloo.ca/~smann/PGABLE/`, 2024.
6. Mann, S., Dorst, L., and Bouma, T. The Making of GABLE, a Geometric Algebra Learning Environment in Matlab. In *Advances in Geometric Algebra with Applications in Science and Engineering*. Editors: E. Bayro-Corrochano and G. Sobczyk. Birkhäuser, 2001.
7. Mann, S., and Leger, Z. Addendum to A Tutorial for Plane-based Geometric Algebra. Available at `https://cs.uwaterloo.ca/~smann/PGABLE/`, 2025.
8. Ong, J. `https://www.jeremyong.com/klein/geometry-potpourri/#the-poincare-dual-map`
9. Manel Velasco, Isiah Zaplana, Arnau Dória-Cerezo, Pau Martí. *Symbolic and User-friendly Geometric Algebra Routines (SUGAR) for Computations in Matlab.* arXiv preprint https://arxiv.org/abs/2403.16634