https://doi.org/10.53656/math2024-1-2-ana *Science in Education* *Научно-методически статии*

AN APPROACH AND A TOOL FOR EUCLIDEAN GEOMETRY

Dr. Boyko Bantchev, Assoc. Prof.
Institute of Mathematics and Informatics — Bulgarian Academy of Sciences

Abstract. Explorers, teachers, and students of geometry of all kinds are now used to applying computer programs in their work. Software of various sorts is available for that. Geometry itself, though of venerable age, has proven to be open to new views and methods. After briefly surveying the known geometry software, we present a double novelty: a vector-based approach to doing geometry, and a respective program tool, aiding in geometric computing and construction, and suitable for producing high quality drawings.

Keywords: vector algebra; Euclidean geometry; geometry software

1. Software for Euclidean geometry

Several kinds of computer programs provide means for constructing geometric objects and performing geometric operations on them.

Mathematical computing environments, such as Maple, MATLAB, or Mathematica, usually provide packages for exploring Euclidean geometry. Typically, a geometry package is a library of procedures for constructing fundamental figures — lines, segments, circles, arcs, etc., and ones for performing basic computations on such objects, e.g. finding points of intersection, angle bisectors, altitudes, tangent lines, and many others. The scope and complexity of the problems that such libraries solve may vary hugely. It is not unusual for a library to provide solutions to computational geometry problems like constructing Delaunay triangulations and Voronoi diagrams.

In the author's experience, the quality of these solutions also varies. One example is Maple's procedure for solving the Apollonius problem, which only tackles a limited version of the problem, and even then accepts only some of the possible configurations of the input objects, and can still produce incorrect answers.

A small group of programs is dedicated to automated proving of geometry theorems, or providing assistance in finding proofs. Some provers are autonomous programs, others combine reasoning capabilities with interactive geometric model building and editing. Given a description of a geometric scene, they can draw it, find locuses, perform animations, and prove or refute propositions about the participating objects. They usually can also export the drawing, reasoning, or both to LaTeX for mathematical publication purposes.

An example of a geometric prover is GCLC.[1] Its input is a procedural description of the respective scene in a specially designed language with commands for direct and dependent construction, point transforms, simple calculations, drawing, annotating, and setting visual attributes. Some provers combine reasoning capabilities with interactive geometric model building and editing.

Yet another kind of programs is concerned solely with constructing geometric objects and scenes of them, finding locuses and animating, based on a description in a specific language. A popular example is ΕΥΚΛΕΙΔΗΣ,[2] whose language is much smaller than e.g. GCLC's, but is nevertheless better structured and more convenient in use.

An important note to make is that, in whatever geometric system, featuring a written construction language is a major advantage. The best way of presenting and communicating — both to humans and to programs — the entire content and the very meaning of a drawing is to write down, in a systematically organized form, a description of the set of actions that produce the drawing's parts, along with the relations between them.

For many years now, researchers, teachers, students and others engaged in mathematical exploration manifest a steady interest in the so called dynamic geometry software — computer programs for interactive creation and manipulation of geometric constructions. Such programs build a geometric model of objects, such as points, lines, circles, etc., together with the dependencies that may relate the objects to each other. The user can manipulate the model by moving some of its parts using a mouse or similar device, and the program accordingly repositions the other parts, so that the constraints are preserved.

Rather than just create images, a drawing in dynamic geometry program is a visualisation of an abstract model of geometric nature and, importantly, provides a visual interface for its manipulation. These programs vary significantly in their drawing capabilities, but they are all centred around geometric modelling. A geometric model may be used to visualize complex geometric constructions, to build and test geometric hypotheses, or to create geometrically precise illustrations to be used in printed documents or on the Web.

Model building in most such systems starts with creating a set of independent, freely existing objects — usually points, and proceeds by constructing ones that are dependent on the former through being geometrically related to them. These programs are mostly used for planar geometry, but a number of them allow for spatial constructions as well.

Some of the widely used dynamic geometry programs are The Geometer's Sketchpad, Cabri, GeoGebra, C.a.R., Cinderella, Kig, and MathKit (a program whose original name is in Russian: Математический конструктор).

An interesting variation of the dynamic geometry approach is presented by the Geometry Expressions program. Rather than being construction-based — proceeding from free to dependent objects, and further to ones that depend on the dependent — the geometry models created by this program are constraint-based. A model is specified by drawing a sketch — an inaccurate representation — and then attaching symbolic constraints and dependencies to it. In order to rebuild the drawing according to the constraints, the program internally creates a construction sequence for the model and executes it, supplying sample numerical values for the free variables remaining in the definition.

Some of the dynamic geometry systems offer a construction language: each command to the system can be entered in a text form, along with its arguments. In addition to the general note we made above on the desirability of a construction language, with respect to systems with a graphical user interface there are more considerations in favour of such a language. One is that, if commands can be nested, this can help avoid cluttering the geometric model with de facto unnecessary names. For example, a command such as

$$\mathbf{line}(\mathbf{intersect}(a,b),\mathbf{intersect}(c,d))$$

would construct a line through the points of intersection of lines a and b, and c and d, without having to name these points.

Even more importantly, while providing all the functionality of a system only through a graphical user interface is either impossible or exceedingly cumbersome, a construction language can comprehensively represent command sets of any amount and complexity.

It also helps a lot if one can input at once a batch of commands, designed and written in advance, and possibly do that more than once in the course of a working session. Otherwise, the usefulness of the command language is limited.

A geometry system can go together with a language that not only handles construction but has means for general programming, such as conditional and repeated execution and user-defined procedures. The language is then called a scripting language, and in fact makes the system programmable. The specific ways in which scripts can be used and interact with each other add to the usability of a programmable system.

GeoGebra is an example of a geometric system, featuring only a construction language: a set of commands for object construction, assigning attributes, setting modes and some others. There is only a limited way for storing and executing commands in groups. Cinderella, on the other hand, sports full programmability through its language CindyScript.[3]

Finally, ability for program-assisted geometric exploration can also be provided in the form of a program library, as is the case with JSXgraph.[4] It is a set of procedures and conventions for using them that constitute a platform for geometric programming in a browser environment. Interactive programs, written in the JavaScript language and executing in a browser, can make use of JSXgraph for solving geometric problems. Besides visualizing a geometric scene, the library supports dynamic geometry by distinguishing between free and dependent objects. Graphical means for interacting with a geometric scene are built-in in the visual box that represents the scene, and there can be as many such boxes as necessary, each holding a ‘live’ scene with draggable objects.

2. What is the proper calculation language for geometry?

To the extent that a geometry system supports calculations that can produce geometric figures and operate on them, they are usually performed on coordinates, most often Cartesian ones. Vectors and complex numbers may also be available. These are the means of expression that we know mostly from analytic geometry. The same tools are in use in hand calculations and in computer-performed computations within geometry programs.

However, having to express all properties and relations of geometric objects in terms of coordinates is too often too tedious, and one can easily end up with a barely readable mesh of equations, hard to comprehend and maintain. Coordinates have no geometric meaning themselves, and the same is true of many arithmetic expressions with coordinates that one has to deal with. Most importantly, if geometric objects are represented in coordinates, then what one works with is not these objects but, rather, a tightly entangled mix of them and the coordinate system with its own peculiarities. And it may be far from obvious what geometric object, if any, is represented by a certain expression in coordinates, and what of its properties, observable through this expression, are really its own, and which ones can be attributed to its representation in coordinates.

Things that we might have expected to be expressed simply, in coordinates may turn to be grotesquely large and complicated. For example, how is one to recognize

$$\left(\frac{(x_B-x_A)\,(x_D y_C-x_C y_D)-(x_D-x_C)\,(x_B y_A-x_A y_B)}{(x_B-x_A)\,(y_C-y_D)-(x_D-x_C)\,(y_A-y_B)},\right.$$
$$\left.\frac{(x_B y_A-x_A y_B)\,(y_C-y_D)-(x_D y_C-x_C y_D)\,(y_A-y_B)}{(x_B-x_A)\,(y_C-y_D)-(x_D-x_C)\,(y_A-y_B)}\right)$$

as the point of intersection of the lines AB and CD? Or, given the above, how easy is it to ascertain that the point belongs to both lines? Or at least to one of them? Surely a nasty heap of variables, yet this is what we obtain in terms of coordinates, given those of A, B, C, and D.

For several reasons that we are not going to discuss here, complex numbers score no better as a *practical* tool of geometric calculations. The following is the same point of intersection, all points being represented by complex numbers:

$$\frac{(c\bar{d}-\bar{c}d)(b-a)-(a\bar{b}-\bar{a}b)(d-c)}{(b-a)(\bar{d}-\bar{c})-(\bar{b}-\bar{a})(d-c)}\,.$$

Although to each point corresponds a single number, and not a pair of coordinates, there are twenty (!) references of names in the above expression — each name had to be used five times. This is hardly the dreamed language to deal with calculations in geometry.

In the next section we make the point that vector algebra is a truly expressive and practical tool for handling calculations in Euclidean geometry. We specifically consider plane geometry. As it turns out, calculating with vectors is remarkably simple and concise. To achieve this, to the well known operations on vectors we only have to add two. They too have been known since the very invention of vectors but were mostly neglected.

3. Planar vector algebra and geometry

Vectors were invented, or perhaps we should say discovered, because geometers and physicists using mathematical methods wanted to be able to assign orientation to certain objects — e.g. segments, lines, planes and surfaces in geometry. Perhaps because of the physicists, to whom the plane was of no interest, more energy and enthusiasm was invested in the development of oriented geometry in space. But then something strange happened. After much of a heated debate in the second half of the 19-th century on the precise formalism and notation to be used in calculating with oriented objects in space, when the version of Gibbs and Heaviside finally took over, its development stagnated, and its use, in geometry at least, remained rudimentary. Mathematicians were so fascinated by the application of matrices and determinants

— also newly invented back then but essentially based on calculating with coordinates — that they developed the habit of not relying on vectors as a language of its own virtue, and only use them as a vehicle to get to matrices and coordinates. In this respect, witness any textbook on analytic geometry.

As for the vector algebra in the plane, it was never properly developed, although all necessary was in place for that. Correspondingly, the use of vectors in planar geometry even today is next to non-existing (not the least because of those many geometers who in fact oppose all forms of calculation, favouring only synthetic reasoning).

Now let's move on to how we use vectors. In the rest of this section we aim to put together convincing evidence for the utility of the vector algebra as a language for Euclidean geometry. All definitions and results that we reproduce here are either well known at a college level (and easily recognizable as such) or author's own.

First of all, vectors can be added, subtracted, and multiplied by a number, these operations having the usual properties. (Algebraically speaking, vectors form a vector space.)

Of two different and non-opposite directions in the plane, we say that one of them precedes the other when, with respect to a line with the first direction, the second direction points towards the left half-plane.

Since vectors are directed, the precedence relation applies to non-parallel, non-zero vectors, too, and we write $\mathbf{u} \prec \mathbf{v}$ when $\mathbf{u}$ precedes $\mathbf{v}$.

We define $\mathbf{u} \times \mathbf{v}$ to be the *oriented area* of the parallelogram built on (representatives of) the vectors $\mathbf{u}$ and $\mathbf{v}$ as its sides. The oriented area is positive when $\mathbf{u} \prec \mathbf{v}$, negative when $\mathbf{v} \prec \mathbf{u}$, and 0 when $\mathbf{u} \| \mathbf{v}$ (i.e. when the parallelogram is degenerate, including the case when any of $\mathbf{u}$ or $\mathbf{v}$ is $\mathbf{0}$).

Finally, we define $\mathbf{u}^\perp$ to be the vector the same length as $\mathbf{u}$, whose direction is at a right angle to that of $\mathbf{u}$, and such that $\mathbf{u} \prec \mathbf{u}^\perp$. Also by definition, $\mathbf{0}^\perp = \mathbf{0}$.

We call $\mathbf{u} \times \mathbf{v}$ the *area product* of $\mathbf{u}$ and $\mathbf{v}$, and $\mathbf{u}^\perp$ the *'perp'* of $\mathbf{u}$ (fig. 1). Clearly, $\times$ and $^\perp$ are specific to vectors in the plane. We purposely use $\times$ to denote area product, as this operation has much in common with the vector product of spatial vectors, denoted the same way.

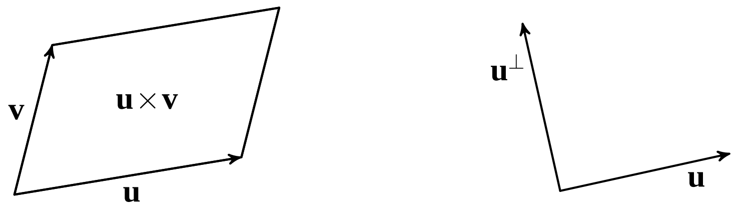

Figure 1. Area product and 'perp'

Of course, we also use the scalar product of vectors, $\mathbf{u}\cdot\mathbf{v}$, defined as known from elsewhere. Note that, since $\mathbf{u}\cdot\mathbf{v}=\mathbf{u}\times\mathbf{v}^{\perp}$, each of the area and scalar products can be defined in terms of the other. The area product is more fundamental, though, because it does not rely on the notion of length (of e.g. vector) — in fact, it can be defined purely algebraically, with no relation to area or geometry at all.

The two products are complementary to each other in that, just as the zeroness or the sign of $\mathbf{u}\times\mathbf{v}$ is an algebraic test for concurrency or precedence, the zeroness or the sign of $\mathbf{u}\cdot\mathbf{v}$ is an algebraic test for perpendicularity, acute or obtuse angle between directions.

Variants of the area product and the perp operation have been known for about 180 years, but rarely, if at all, received due attention. For the usefulness of the vector algebra in the plane, and for the breadth of its application to geometry, these two operations are vital.

Some identities involving $\cdot$, $\times$, and $^{\perp}$ are (note that in $\sin(\mathbf{u},\mathbf{v})$ the angle is oriented from $\mathbf{u}$ to $\mathbf{v}$):

$$\begin{aligned}
\mathbf{u}\cdot\mathbf{u} &= |\mathbf{u}|^2\\
\mathbf{u}\cdot\mathbf{v} &= \mathbf{v}\cdot\mathbf{u}\\
(k\,\mathbf{u}+k'\,\mathbf{v})\cdot\mathbf{w} &= k\,(\mathbf{u}\cdot\mathbf{w})+k'(\mathbf{v}\cdot\mathbf{w})\\
\mathbf{u}\cdot\mathbf{v} &= |\mathbf{u}||\mathbf{v}|\cos(\mathbf{u},\mathbf{v})\\
|\mathbf{u}\cdot\mathbf{v}| &\le |\mathbf{u}||\mathbf{v}|\\
(\mathbf{u}^{\perp})^{\perp} &= -\mathbf{u}\\
(k\,\mathbf{u}+k'\,\mathbf{v})^{\perp} &= k\,\mathbf{u}^{\perp}+k'\,\mathbf{v}^{\perp}\\
\mathbf{u}^{\perp}\cdot\mathbf{v} &= -(\mathbf{u}\cdot\mathbf{v}^{\perp})\\
\mathbf{u}^{\perp}\cdot\mathbf{v}^{\perp} &= \mathbf{u}\cdot\mathbf{v}\\
\mathbf{u}\times\mathbf{v} &= -(\mathbf{v}\times\mathbf{u})\\
(k\,\mathbf{u}+k'\,\mathbf{v})\times\mathbf{w} &= k\,(\mathbf{u}\times\mathbf{w})+k'(\mathbf{v}\times\mathbf{w})\\
\mathbf{u}\times\mathbf{v} &= |\mathbf{u}||\mathbf{v}|\sin(\mathbf{u},\mathbf{v})\\
|\mathbf{u}\times\mathbf{v}| &\le |\mathbf{u}||\mathbf{v}|\\
\mathbf{u}^{\perp}\times\mathbf{v} &= -(\mathbf{u}\times\mathbf{v}^{\perp})\\
\mathbf{u}^{\perp}\times\mathbf{v}^{\perp} &= \mathbf{u}\times\mathbf{v}\\
\mathbf{u}\times\mathbf{v} &= \mathbf{u}^{\perp}\cdot\mathbf{v}
\end{aligned}$$

Many other identities hold as well, filling in the content of planar vector algebra and useful for obtaining further results. Here are several such identities, for any vectors **a**, **b**, **c**, and **d** in the plane:

$$(\mathbf{b}\times\mathbf{c})\,\mathbf{a}+(\mathbf{c}\times\mathbf{a})\,\mathbf{b}+(\mathbf{a}\times\mathbf{b})\,\mathbf{c} \;=\; \mathbf{0}$$

$$(\mathbf{a}\times\mathbf{b})^2+(\mathbf{a}\cdot\mathbf{b})^2 \;=\; \mathbf{a}^2\,\mathbf{b}^2$$

$$(\mathbf{a}\times\mathbf{c})\,\mathbf{b}^2 \;=\; (\mathbf{a}\times\mathbf{b})(\mathbf{b}\cdot\mathbf{c})+(\mathbf{a}\cdot\mathbf{b})(\mathbf{b}\times\mathbf{c})$$

$$(\mathbf{a}\cdot\mathbf{c})\,\mathbf{b}^2 \;=\; (\mathbf{a}\cdot\mathbf{b})(\mathbf{b}\cdot\mathbf{c})-(\mathbf{a}\times\mathbf{b})(\mathbf{b}\times\mathbf{c})$$

$$\begin{aligned}(\mathbf{a}\times\mathbf{b})(\mathbf{c}\times\mathbf{d}) &= (\mathbf{a}\times\mathbf{c})(\mathbf{b}\times\mathbf{d})-(\mathbf{b}\times\mathbf{c})(\mathbf{a}\times\mathbf{d})\\ &= (\mathbf{a}\cdot\mathbf{c})(\mathbf{b}\cdot\mathbf{d})-(\mathbf{b}\cdot\mathbf{c})(\mathbf{a}\cdot\mathbf{d})\end{aligned}$$

Behind some such identities one can easily spot well known trigonometric ones. But the vector identities are being established with no direct reference to angles at all. As vectors possess direction, operations on vectors have the inherent ability to perform various implicit computations with angles. It is very characteristic of vector algebra that with it there is hardly a need for trigonometry — the latter is simply superseded by the more general and arguably more natural to geometry language of vectors.

Of particular importance is the following fact: if $\mathbf{u}\times\mathbf{v}\neq 0$ (i.e. **u** and **v** are non-zero and non-parallel), any vector **p** admits a unique decomposition along **u** and **v** (fig. 2), namely

$$\mathbf{p} \;=\; \frac{\mathbf{p}\times\mathbf{v}}{\mathbf{u}\times\mathbf{v}}\,\mathbf{u}+\frac{\mathbf{u}\times\mathbf{p}}{\mathbf{u}\times\mathbf{v}}\,\mathbf{v}.$$

Specifically for $\mathbf{v}=\mathbf{u}^{\perp}$:

$$\mathbf{p}=(\hat{\mathbf{u}}\cdot\mathbf{p})\,\hat{\mathbf{u}}+(\hat{\mathbf{u}}\times\mathbf{p})\,\hat{\mathbf{u}}^{\perp}.$$

(Here and elsewhere, $\hat{}$ denotes a unit vector with the direction of a given vector.)

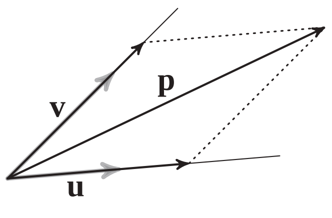

Figure 2. Vector decomposition

It appears that a great number of vector-related and geometric results follow directly from the decomposition formula.

For example, the oriented distance of a point P to a line passing through a point A and having the direction of a vector **u** is $\hat{\mathbf{u}}\times\mathbf{AP}$ — positive when P is on the left of the line. The position vector of the projection of P on the line is $\mathbf{A}+(\hat{\mathbf{u}}\cdot\mathbf{AP})\hat{\mathbf{u}}$. We can read out both these from the two parts in which **AP** decomposes along $\hat{\mathbf{u}}$ and $\hat{\mathbf{u}}^{\perp}$.

Furthermore, the equation

$$s\,\mathbf{p}+t\,\mathbf{p}^{\perp}=\mathbf{u},$$

can be solved for **p**, obtaining

$$\mathbf{p}\;=\;\frac{s\,\mathbf{u}-t\,\mathbf{u}^{\perp}}{s^2+t^2}\cdot$$

Similarly, the solutions of

$$\mathbf{u}\times\mathbf{p}\;=\;c\qquad\text{and}\qquad\mathbf{u}\cdot\mathbf{p}\;=\;c\tag{1}$$

are

$$\mathbf{p}\;=\;s\mathbf{u}+\frac{c}{\mathbf{u}^2}\mathbf{u}^{\perp}\qquad\text{and}\qquad\mathbf{p}\;=\;s\mathbf{u}^{\perp}+\frac{c}{\mathbf{u}^2}\mathbf{u}\tag{2}$$

for any number s.

Solving two such equations simultaneously, we obtain a single vector. Thus, the solution of e.g.

$$\begin{aligned}\mathbf{u}\times\mathbf{p}\;&=\;s\\ \mathbf{v}\times\mathbf{p}\;&=\;t\end{aligned}\tag{3}$$

given $\mathbf{u}\times\mathbf{v}\neq 0$ is

$$\mathbf{p}\;=\;\frac{s\,\mathbf{v}-t\,\mathbf{u}}{\mathbf{u}\times\mathbf{v}}\cdot\tag{4}$$

An equation of any of the two kinds (1) admits an immediate interpretation as an equation of a straight line if **p** is considered the position vector of a variable point. The line is either parallel or perpendicular to **u**, depending on whether the multiplication is $\times$ or $\cdot$. Correspondingly, any of (2) can be seen as the parametric equation of this line. And solving a system of two equations, each of the kind (1) or (2), is none else but finding the point of intersection of the respective lines.

For example, if A abd B are two points, a point P is of the line AB if and only if

$$\mathbf{AB}\times\mathbf{AP}\;=\;0.$$

This is a remarkably simple equation of a line by two known points, as only a single algebraic operation is involved! By substituting $\mathbf{AP}=\mathbf{P}-\mathbf{A}$ and $\mathbf{AB}=\mathbf{B}-\mathbf{A}$ the equation takes the form $\mathbf{AB}\times\mathbf{P}=-\mathbf{A}\times\mathbf{B}$, which is of the kind (1). Given another line CD and proceeding similarly, we obtain a system of the kind (3), wherefrom, applying (4), we obtain the position vector of $AB\cap CD$:

$$\mathbf{P} \;=\; \frac{(\mathbf{C}\times\mathbf{D})\,\mathbf{AB}-(\mathbf{A}\times\mathbf{B})\,\mathbf{CD}}{\mathbf{AB}\times\mathbf{CD}}\,.$$

Or, we could have kept the first equation in the form $\mathbf{AB}\times\mathbf{AP}=0$, transformed the second one to $\mathbf{CD}\times\mathbf{AP}=\mathbf{CD}\times\mathbf{AC}$, and solved for $\mathbf{AP}$:

$$\mathbf{AP} \;=\; \frac{\mathbf{AC}\times\mathbf{AD}}{\mathbf{AB}\times\mathbf{CD}}\,\mathbf{AB}$$

(which also follows directly from the already found result for $\mathbf{P}$ above).

Thus we see that, by making use of the algebra of vectors, lines and anything related to them can be expressed and worked out in terms of the points and directions that define the lines. This applies to other geometric figures as well.

If the notation $[P_1P_2\,...\,P_n]$ be adopted for twice the oriented area of the polygon $P_1P_2\,...\,P_n$, many of the expressions involving $\times$ can be rephrased in terms of oriented areas of polygons. This, too, is very much in the spirit of how we apply vector algebra to geometry: vectors are a language and means, not an end in itself.

For example, as $\mathbf{AC}\times\mathbf{AD}=[ACD]$ and $\mathbf{AB}\times\mathbf{CD}=[ACBD]$, the above expression for $\mathbf{AP}$ obtains a form with only a minimal presence of vectors:

$$\mathbf{AP} \;=\; \frac{[ACD]}{[ACBD]}\,\mathbf{AB}\,.$$

This form exhibits utmost simplicity, but the relation expressed in it is not trivial, taking into account that $ACBD$ can be any quadrilateral, including self-intersecting. It is rather unlikely that such a formula could have been arrived at without using vector algebra. In our practice of calculating with vectors, we have often encountered this phenomenon.

One use of having a general formula for finding points of intersection is to find expressions of triangle centres in terms of vertices and sides. For example, letting $\mathbf{a}=\mathbf{BC}$, $\mathbf{b}=\mathbf{CA}$, and $\mathbf{c}=\mathbf{AB}$ in $\triangle ABC$, the altitude through, say, A has the equation $\mathbf{a}\cdot\mathbf{AP}=0$, or, equivalently, $\mathbf{a}^{\perp}\times\mathbf{AP}=0$. The orthocentre H then can be found as the point of intersection of any two of the altitudes:

$$\mathbf{H} \;=\; -\frac{((\mathbf{A}\cdot\mathbf{a})\,\mathbf{A}+(\mathbf{B}\cdot\mathbf{b})\,\mathbf{B}+(\mathbf{C}\cdot\mathbf{c})\,\mathbf{C})^{\perp}}{[ABC]}\,,$$

and $\mathbf{AH} = \dfrac{\mathbf{b}\cdot\mathbf{c}}{\mathbf{b}\times\mathbf{c}}\,\mathbf{a}^{\perp}$ also holds.

As an example related to a non-linear geometric figure, here is an equation of the disk whose circumference passes through points A, B, and C:

$$((\mathbf{CA}\cdot\mathbf{CB})\,(\mathbf{PA}\times\mathbf{PB}) - (\mathbf{CA}\times\mathbf{CB})\,(\mathbf{PA}\cdot\mathbf{PB}))\,\mathrm{sign}[ABC] \;\geq\; 0\,,$$

where $\geq$ is actually $=$ or $>$ for P on or strictly inside the circumference.

Equations of many curves can also be expressed in a vector language. For example, conics have the general equation

$$\mathbf{FP}^2 = ((e{+}1)\,|\mathbf{r}| - e\,(\hat{\mathbf{r}}\cdot\mathbf{FP}))^2,$$

where e, $\mathbf{r}$, and F determine the curve's main characteristics, namely (see fig. 3):

e	eccentricity — effectively type (circle, ellipse, hyperbola, or parabola) and specific shape;
$\lvert\mathbf{r}\rvert$	scale (focus-to-apex distance);
$\hat{\mathbf{r}}$	orientation (of the axis, focus-to-apex);
F	position (of the focus).

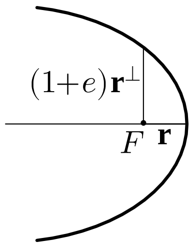

Figure 3. Defining a conic

Contrary to the popular belief, conics and related matters can be treated much more properly with vectors than with coordinates.

Many kinds of geometric transforms, and a number of other topics in plane geometry, are handled most adequately in the vector language. It is equally well suited for the proving and the construction type of problems. As vectors serve successfully both the classical and the analytic realms of geometry, they can be seen as a unifying language for the discipline. In (Bantchev 2017) and elsewhere, we have outlined some advantages of the calculational approach, specifically the one based on vectors, over the synthetic one in geometry.

4. A language for geometric programming

Having an extensive and gratifying experience with developing the two-dimensional vector algebra and its application to plane geometry, a natural next step was to consider casting this approach into a language that could be used on a computer. Doing calculations, deriving formulae, and describing geometric figures in terms of other figures was already a form of programming, but it remained to give the algebraic language an executable, computing-oriented shape, ensuring practicability.

Let us sketch the rationale and the main features of the language we designed with the above in mind.

Unlike the languages in the known geometry systems or libraries mentioned in section 1, our language does not aim to provide 'high-level' constructions of geometric figures. Most such constructions, as found in the command menus of a dynamic geometry program or in a geometric language, boil down to a line or two when expressed as calculation with vectors on the given data. Tens of such constructions can be encoded as a short list of formulae.

Therefore, by providing a useful set of operations on vectors, our language ensures utmost versatility while remaining small. In place of a fixed list of geometric constructions that, no matter how large, cannot meet all possible needs, there is a small yet powerful set of basic operations, out of which any conceivable geometric construction can be built, and succinctly at that. This approach easily achieves what in a closed and finished form is present elsewhere, and much more than that.

From the viewpoint of a teacher or a learner of geometry, the said approach also has the virtue of keeping in observable, text form all needed geometric constructions. They can thus be studied, compared, edited, reused, etc.

It must be stressed that practicability of the language is achieved through broad use of vector operations. Computing with coordinates is supported but very rarely needed.

On the other hand, we deemed necessary to augment the small set of vector operations presented in section 3. These operations suffice to express any computation, but for practical purposes it is useful to have shortcuts for different ways of producing vectors or numbers out of other vectors or numbers. That is why such shortcuts were included in the language. We describe them in the next section, along with many other operations.

Besides numbers and vectors, other kinds of data — Booleans, text strings, and sequences — were also considered useful for the language, and thereof included in it.

Text strings may seem to have little use in geometric computation, but sometimes one needs to properly format a number, and perhaps to put together several pieces of text, so that the result of a computation is output in a readable form. Strings also serve as keywords designating construction sub-commands and other things related to creating a drawing.

Sequences can be formed of values of any kind, including other sequences, and are thus hierarchical structures. They can be used to represent geometric figures of any sort and complexity. For example, a line segment may be represented as a pair of two end points (position vectors), a polygon — as any number of vertices, etc. Obviously, keeping a flat or hierarchical scene as a sequence is also possible, and in more than one way.

Sequences play an important role in the part of the language responsible for drawing.

There are no points as a separate kind of values in the language. There is no need for them either, because computationally a point is indistinguishable from its position vector.

At present, the calculational part of the language consists of nearly eighty operations in twelve levels of relative precedence. A small number of operations have names made of letters, and the rest are designated each by a single non-alphabetic character. This is achieved by a heavy use of name overloading: using a single name for several (as it turns out, up to eight) operations, the distinction among them being made by the types of the arguments used in each single instance of an operator.

This is not unlike what one finds in programming languages such as APL and J, although the actual mapping of names to meanings is very different. The overloading is a result of very carefully choosing the names so that, in making sense of them, a reader is aided by ‘naturalness’ and semantic affinity. For example, the same character (`-`) is used for arithmetic and logical negation, string and sequence reversal, and reversal of (the direction of) a vector, all of which have the idea of oppositeness in common. Similarly, the character `|` is used for any of absolute value of a number, length of a vector, and the number of elements in a text string or a sequence, all these ultimately being magnitudes.

The language admits introducing variables, along with certain freedom in naming them. The name of a variable may be any mixture of Latin, Greek, and Cyrillic letters, `'` (quote), decimal digits, as well as the subscript digits $_{0123456789}$. Examples of names are φ, `x'`, a_4, and λμν`'xy`.

Some of the operations have aliases, to be used depending on the intended meaning in a particular use, or simply on preference.

An important characteristic of the language is that it strictly separates computation from presentation. There are in fact two co-existing languages: one that is concerned with computing and possibly producing abstract geometric constructions, and another that enables the production of a graphical image of a geometric scene. The latter sub-language consists of a single command, `#show`, the argument to which is a sequence. The sequence is structured according to simple rules, but can be arbitrarily complex, representing anything from a single geometric object to a scene composed of many such objects with different visual attributes.

To each geometric object or a group of objects can be attached colour, line width, and other visual attributes, as well as a linear transformation. Nesting groups of objects is obtained through nesting sequences, and in the thus formed hierarchy attributes apply locally at the level to which they are attached. Nested sequences can also be used as layers: sets of related objects that are to be added to the scene or removed from it as one whole.

Separating computation from drawing has several implications which deserve to be emphasized as differences from most geometric systems, where an object is drawn implicitly upon construction, and tacit assumptions affect actual drawing.

First, constructing a geometric object in our language is merely computing its defining data — say, the centre and radius of a circle. It has nothing to do with whether the object is going to be drawn, and what visual attributes apply if it is. Second, giving a name to an object is done for the sole purpose of making it possible to refer to that object. It does not mean that the same name should accompany the object if the latter is drawn. Labels and denotations attached to the drawing are entirely independent of any names possessed by the objects in a script, and are placed explicitly, precisely where they are wanted. Third, the order in which objects are placed in the drawing is unrelated to the order in which objects are being constructed. This is rather important as the former order defines the visual overlapping between objects, hence it must be chosen freely as wished or needed.

5. Operators on vectors and other data

In this section we glance at the many operations available in the language for computing with numbers, vectors and other values.

The four basic arithmetic operations on numbers are designated by the usual characters `+`, `-`, `*`, and `/`. There is also `^` — exponentiation, and `%` — remainder of division in which the quotient, if not an integer, is replaced by the nearest number below it (so that e.g. `13%-5` would produce -2). There are three trigonometric operations, namely `#sin`, `#cos`, and `#tan`.

Numbers can be compared with `<`, `>`, `=`, or `~` (not equal). All comparisons are performed up to a tolerance, which is set as the value of the special variable `#eps`. For chosing the maximum and minimum of two numbers, there are `|` and `&`, respectively, which are also the ‘logical or’ and ‘logical and’ of Booleans. Booleans too can be compared. They can also be used as arguments to operations that expect a numeric argument, 'true' then meaning 1 and 'false' meaning 0.

Provided are two numeric constants — `#pi`, which is π, and the base of natural logarithms `#e` — and the two Boolean constants, `#t` and `#f`.

Most mentioned names also designate one-argument operations as follows. Arithmetic and Boolean negation are both named `-`, the tilde `~` is 'not zero', `*` is the sign operation (-1, 0, or 1, according as the argument is negative, zero, or positive), `/` gives the reciprocal of a number, `|` the magnitude ($|\dots|$), and `^` the natural logarithm of the argument. `<`, `>`, and `=`, when used as monadic operations, round their argument to an integer, if it is not already one: the nearest below, the nearest above, and the result of truncating the fractional part. Thus, `<-2.7` would produce -3, while any of `>-2.7` and `=-2.7` gives -2.

Naturally, many operations in the language act on vectors, produce vectors, or do both of these. For example, a vector can be constructed in several ways: from coordinates, as in `5,3.6`; from a length and direction angle; from direction angle only (a unit vector); by rotating a vector to an angle; by changing the length of another vector. The last four operations are all denoted `@`, but no ambiguity arises with respect to which of them acts in a particular instance, as the arguments differ in number and type.

The most basic arithmetic operations on vectors are the same as for numbers, e.g. `-` is both subtraction of vectors and opposite vector, and the monadic `|` finds the length (i.e. magnitude) of a vector. For the direction of a vector and the angle between two vectors, again monadic and dyadic instances of `@` are used. The `<` and `>` comparisons are, like with numbers, up to tolerance, but are used to test for precedence of directions ($\prec$ or $\succ$) of the respective vectors. Another operation (dyadic `|`) tests for concurrency.

Yet another way to construct a vector is by reverse subtracting. For instance, if `A` and `B` are vectors, `A_B` is the same as `B-A`. The arguments of `_` can be any expressions, but this operation is most useful with names — it is meant to imitate the widely used mathematical notation: **AB** as a shortcut for $\mathbf{B}-\mathbf{A}$. For the same reason `_` has the highest precedence among all operations, so e.g. `A_B^A_C` correctly computes the area $[ABC]$.

For multiplication, the character `·` may be used as an alias of `*`, which of course is useful for visually distinguishing scalar products. As for the area product and the $\perp$ operation, `×`, `^`, and `⊥` are interchangeable, both in the monadic and the dyadic cases. The monadic case also applies to any sequence $P_1P_2\dots P_n$ of (position) vectors, computing $P_1{\times}P_2+\dots+P_{k-1}{\times}P_k+P_k{\times}P_1$, which is $[P_1P_2\dots P_n]$ when $P_1P_2\dots P_n$ is a non-intersecting polygon.

Complex arithmetic is covered in the language by providing operations that treat vectors as complex numbers.

Text strings and sequences, both being linear structures, are handled by more or less the same set of operations. These include finding the length (`|`), a prefix or suffix (`%`), an element with specific index (`/`), reversal (`-`), and catenation (`+`).

Besides catenation, a sequence can be extended by appending another sequence as a single element to it. Thus, if `A` and `B` are sequences of 3 and 5 elements, `A+B` produces a sequence of length 8, made up of all the members of `A` and `B`, while `A,B` has only 4 items. In fact, `B` can be anything: sequence or not, it is a single item in `A,B` at its last position.

Note that with the exception of the empty sequence `()`, there is no special syntax in the language for representing vectors and sequences. A vector or sequence is always produced by applying an operation on suitable arguments, as described above.

The `|` operation, when acting on a number and a sequence of a special kind, produces a string with a formatted representation of the number. For example, the expression `(355/113-#pi) | (10,'(.),15,'(>␣))` would produce the string `␣␣␣0.0000002668` of length 15, in which the number $355/113-\pi$ is represented with 10 digits in the fractional part and is right-aligned within the length of 15, with leading visual spaces.

6. Drawing figures and text

In our geometric language, graphical output is produced by using the command `#show`. To draw a particular geometric figure or a text, one has to fill in a construction sequence of a specific form. There are, at present, eight types of construction sequences: for points, line segments, infinite lines, rays, circles, circular arcs, chains (of line segments and arcs), and text. Any such sequence can be feeded as an argument to `#show`.

Each construction sequence starts with a string that tells what kind of object is to be constructed — `"point`, `"line`, `"iline`, `"ray`, `"circle`, `"arc`, `"chain`, or `"label` — followed by one or more arguments, specific for the object.

More specifically, a point construction requires a (position) vector, and a line segment or a line requires two such vectors. Similarly for a ray, but there one of the vectors specifies a direction rather than a point. A circle is specified by a position vector for the centre and a number for the radius.

For arcs, there are two options. One of them is to specify the arc by its centre, start point, and angular span. The other option requires start and end points, along with another point between them and also on the arc.

Yet another option for constructing a circular arc is available through building a chain, where, along with a start and end points, an oriented distance of the centre to the chord is given, together with arcs orientation, positive (CCW) or negative. A chain is a sequence of line segments and arcs, and as a line may be open or closed.

Drawing a ray goes together with producing an arrow mark in the drawing which visually informs of the respective direction. An arrow mark can also be attached to a line segment, an infinite line, or an arc by providing an optional argument at the end of the respective construction sequence — a number which defines where exactly along the said object the arrow mark must be placed.

With a `"label` construction sequence, a drawing can be annotated with names of objects and other helpful text. Graphical attributes, such as scale and colour, apply to text annotations just as they do to any geometric object.

A useful technique when placing a piece of text with `"label` is to separate the position into absolute and relative parts. The former is typically a point, and the latter an offset from that point. For instance, with the sequence

```
"label,'(β/2),K+2@-15
```

the text $\beta/2$ is put at a distance 2 in the direction of -15° from point `K`. This is done by adding the vectors `K` and `2@-15`, the latter of which happens to be computed in polar form.

Specific attributes can be attached to one or more construction sequences, in order to visually distinguish the respective parts from the rest of the drawing or apply a geometric transformation to these parts. Line width and line type, as well as colour and opacity — of a line itself or of the area to which it is a border — can be set as necessary. Translation, rotation and scaling can be applied instead of, or together with, setting visual attributes. By composing these three any affine transform can be obtained.

In general, an argument to `#show` may be any list of items, to which a list of attribute settings is attached. Any of these items can itself be another such list with attributes attached, and so on downwards, ending with construction sequences for points, lines, arcs, and text, as enumerated above. This hierarchy of geometric and text objects comprises the scene to be visualized by `#show`. The higher in the hierarchy, the earlier an object is put on the drawing. Because objects that are put later may partially or entirely hide the ones already placed, this must be taken into account when composing the scene.

Down the hierarchy each attribute remains in effect until another of the same kind appears. At this point transform attributes add up, while graphical ones, such as e.g. colour, get replaced in the local scope.

7. Computer implementation

The language described in the previous three sections has received the name ForGe and has been implemented by the author. The implementation is an interpreter written in the programming language JavaScript and can be run in a Web browser. It is publicly accessible through a simple visual interface from the author's Web page.[5] A series of examples in construction and drawing is available for perusal and experimentation at the same place.

That JavaScript is universally available makes the ForGe implementation independent of the operating system and other elements of the operating environment. In addition, as ForGe scripts are plain text, it is easy to generate them programmatically, or to process them using general text processing tools. Programmatically generated scripts can make use of the whole ForGe language or, say, only of its ability to produce a graphical output. These are important indicators of the ability of the ForGe implementation to co-operate with other software.

As a ForGe script is run in the interpreter, possible syntax and other errors are properly detected and signaled to the user. If correct, the script can produce two kinds of output — text and graphics. The text output consists of ForGe values in a text form. Sequences, being composite values, are output in a structured way, so that their components are clearly distinguished and nesting is visually apparent.

Quality of graphical output is ensured by presenting the drawing in the SVG format. The drawing is immediately seen in the browser, and can be extracted as an SVG file, thus enabling its further use for what it may be needed. As all modern Web browsers, most word processing and document preparation systems, and many other programs can interpret SVG files, the latter are universally usable in all kinds of Web browser hosted environments, as well as in printed documents. The scalability of SVG and its support of a wealth of graphics features make it an excellent choice wherever rich content, high-quality graphics is required.

The SVG that is automatically produced by various programs is often needlessly large, but this is not the case with ForGe — its interpreter is particularly good at producing a space-efficient SVG.

The very interpreter is remarkably small — hundreds to thousands of times smaller than the implementation of any geometric system or library we know of, being at the same time equal or comparable in functionality.

8. Further development

Several directions of extending ForGe and improving its usability are being envisaged by the author. One of them is to make the implementation run as a console program, rather than just in a browser. This is possible and attractive due to the spread of non-browser JavaScript runtime environments, such as Node.js and Deno, across all popular operating systems. Opening the ForGe interpreter to console mode of execution would enable direct interaction with text processing software through channels, pipes and other facilities of the operating system. Both ForGe's input and output can be directed to other programs within a console command or a script of the operating system.

Another direction of improvement is redoing the graphics generation part of the ForGe interpreter so that it can output in other image formats, such as PostScript or PDF, besides SVG. Particularly attractive is generating Ti*k*Z[6] scripts that can be interpreted within TEX, LATEX, and their relatives. In this way drawings can be produced by ForGe, benefiting from its computational machine, and then annotated using the fonts and the very expressive language for composing formulae in a TEX-based system.

The ForGe language itself can be improved in several ways. These include adding conditional and repetitive execution, as well as user-defined procedures. Most constructions in the language are but simple formulae, but even so, referring to a formula by name would be simpler than pasting it each time anew and entirely. Libraries of such definitions can be supplied with ForGe to facilitate creating scripts in it and make them more readable.

Drawing conic sections and providing basic operations on them is another possible enhancement of ForGe.

The current browser-based user interface to ForGe, although effective, is rather limited. There are numerous possibilities of improving it. Building and experimenting with user interfaces, browser-based or other, that provide access to ForGe's geometric computing and drawing, is deemed a very promising field of exploration into both computer geometry and user interfaces in general.

NOTES

1. https://poincare.matf.bg.ac.rs/%7Ejanicic/gclc
2. http://eukleides.org
3. https://doc.cinderella.de/tiki-index.php?page=CindyScript
4. https://jsxgraph.uni-bayreuth.de
5. http://www.math.bas.bg/bantchev/ForGe
6. https://tikz.dev

REFERENCES

BANTCHEV, B., 2017. *Menelaus, Einstein, Peano and the proofs in geometry*. Mathematics and Education in Mathematics: Proc. 46th Spring Conf. of the Union of Bulgarian Mathematicians, vol. 46, pp. 221 – 229.

✉ **Dr. Boyko Bantchev, Assoc. Prof.**
ORCID iD: 0000-0002-0284-312X
Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
Acad. G. Bonchev St., bl. 8
1113 Sofia, Bulgaria
E-mail: boykobb@gmail.com