Title:
Geomatplot: Interactive Geometric Plotting Library

Authors:
Csaba Bálint, csabix@inf.elte.hu, Eötvös Loránd University
Róbert Bán, rob.ban@inf.elte.hu, Eötvös Loránd University

Keywords:
Data Visualization, Interactive Rendering

DOI: 10.14733/cadconfP.2024.222-226

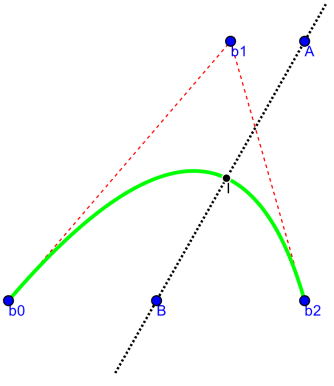
```
clf;                    % Clear figure

A = Point([.9 .9]);     % Draggable points with automatic
B = Point([.5 .2]);     %     labels starting from 'A'
l = Line(A,B,'k:',2); % Line between A and B

b0 = Point('b0',[.1 .2]); % Labeled control points
b1 = Point('b1',[.7 .9]);
b2 = Point('b2',[.9 .2]);
Segment(b0,b1,'r--'); % Segments b0b1 and b1b2 with a
Segment(b1,b2,'r--'); %   short line specification

fun = @(t,b0,b1,b2)  b0.*(1-t).^2 + 2*b1.*t.*(1-t) + b2.*t.^2;
bt = Curve(b0,b1,b2,fun,'g',3);  % Green quadratic Bezier curve
Intersect('I',l,bt);  % Intersection of a curve and a line
```

(a) Functions starting with a capital letter belong to Geomatplot.

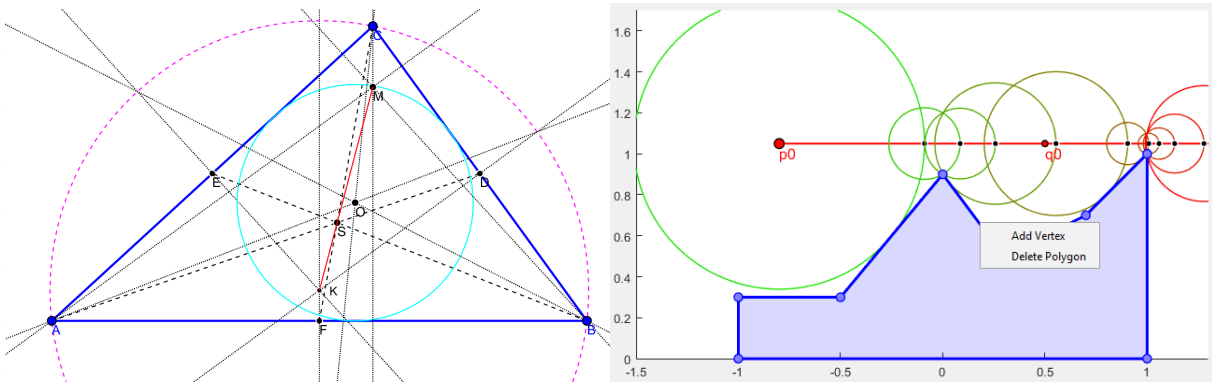


(b) Interactive plot

Fig. 1: A simple MatLab example code and the interactive output plot. Callbacks for the moveable points are created automatically to update the dependent geometries when points are dragged.

Introduction:

Our research in computer graphics and computational geometry frequently necessitated creating various geometric plots for both validation and publication purposes. Our figures and prototypes were created either in Geogebra [1], MatLab [2], or Shadertoy [3] depending on the specific application, however, we found that all such tools are severely lacking in functionality for our use cases. For example, while the interactive aspect of Geogebra is great for figures and formula validation, entering complex formulae, such as curves in various bases, is extremely cumbersome, and hard to edit as there is no global script defining the geometry. Since Geogebra moved to HTML5 and the classic Java implementation supporting Geogebra scripts has been discontinued, the need for a novel, general approach has risen. Moreover, because Geogebra mostly operates with symbolic expressions, certain applets can become unnecessarily slow or unresponsive.



(a) Constructing the Euler line of a triangle (b) Sphere tracing an interactive polygon

Fig. 2: The three blue vertices of the triangle, the p_0 and q_0 points of the red ray, and all the points of the polygon are draggable resulting in a constantly updating plot.

On the other hand, plotting in MatLab or Python can be integrated with existing code allowing for more options, but creating interactive Geogebra-like plots for testing or demonstration requires a great deal of boilerplate coding. Thus, we decided to make our plotting library and engine with the following goals in mind:

1. Easy creation of 2D geometric plots for publication
2. Prototyping computational geometry formulas and algorithms for research
3. Interactive testing of geometric functions by injecting code

These goals require a powerful scripting language with good performance and we opted for MatLab adopting the following design principles to create Geomatplot[1]:

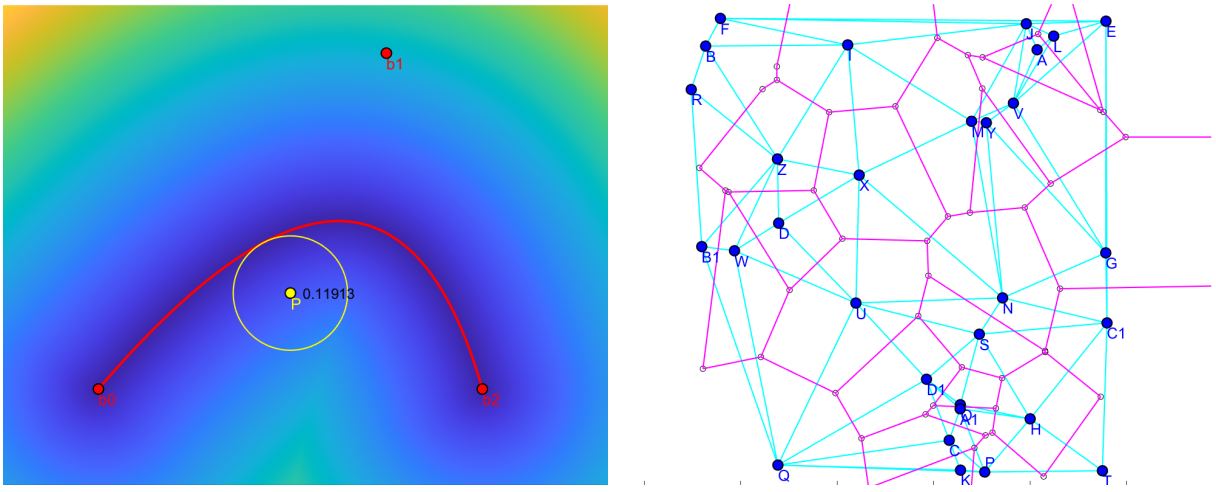
1. Interactive, responsive plots with draggable objects
2. Must be easy to use through scripting with powerful functions
3. Efficient numerical algorithms for fast rendering
4. Allow user-defined callback functions to inject arbitrary algorithms

Usage examples:

Figure 1 showcases an example script with the resulting interactive plot. The black dotted line, the red dashed segments, the green curve, and the black intersection point labeled I will update their position when the blue movable points are dragged. Geomatplot builds an internal dependency graph as the commands are executed on Fig. 1a where each object is either movable or dependent. For example, `A=Point(); B=Point();` creates two movable points, and `Line(A,B)` connects these with a line that depends on `A` and `B`.

The `Curve` command draws a parametric curve that may depend on any number of input geometries, but the user must supply a callback function that takes the $t \in [0, 1]$ parameter and all dependent

[1]Geomatplot available at https://github.com/Csabix/Geomatplot.



(a) Geometry-dependent $\mathbb{R}^2 \to \mathbb{R}$ function

(b) Delaunay triangulation and Voronoi cells

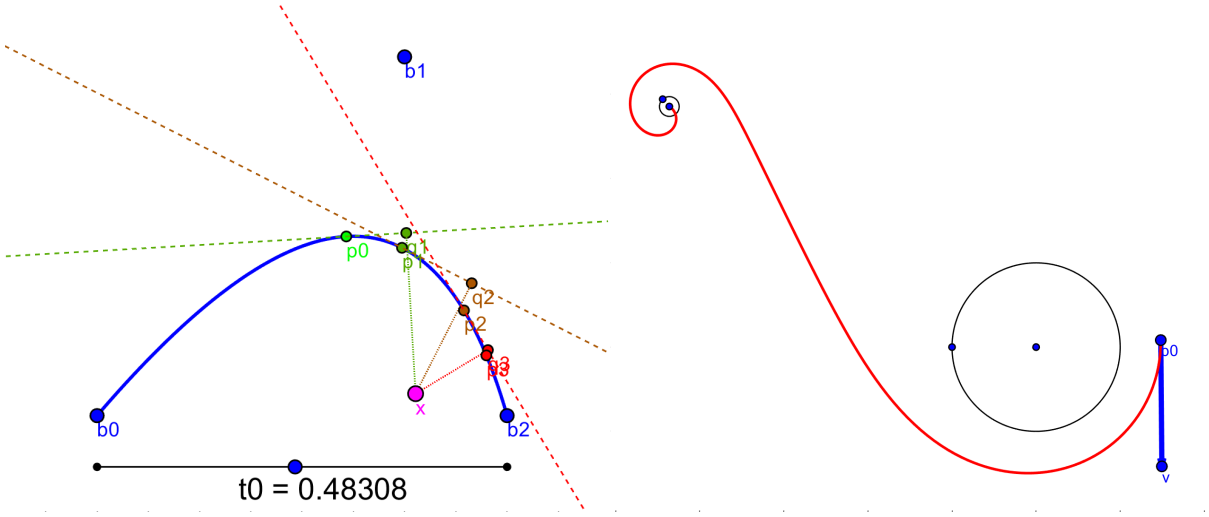
Fig. 3: The `Curve` and the `Image` function plots a user defined $[0, 1] \to \mathbb{R}^2$ curve and $\mathbb{R}^2 \to \mathbb{R}$ image with given dependent input Geomatplot geometries. The `CustomValue` function creates an arbitrary dependent composite object such as a Delaunay triangulation, which then can be used for further plotting.

geometries as arguments. For example, on Fig. 2, our `Circle` command uses built-in callbacks to create such curves for each function overload. The radius of such circles, including the point-to-polygon distance in Fig. 2b, are dependent scalar values created with `Distance` function. Geomatplot overloads operators on these scalars, points, and vectors so these algorithms can be easily defined with simple expressions. For example, a `p=Eval(p0+t*v);` expression is equivalent to `p=Point(p0,t,v,@(p0,t,v)p0+t*v);` call creating the update callbacks automatically.

The `Image` method is similar to `Curve`, but the callback must take two real numbers or one complex number or matrices as inputs, and it can run on the GPU as well. The `Text` function renders dependent text, such as the distance to the curve in Fig. 3a. In Fig. 3b, the built-in MatLab command `delaunayTriangulation` runs in the callback to dynamically create both the Delaunay triangulation and the Voronoi cells for the movable blue points. The `Intersection` command creates one or more intersection points. When the callback of a dependent object fails to define the object properly, for example, an intersection does not exist, the Geomatplot object becomes *undefined*, and all objects depending on it will not be rendered. When an object becomes defined again, the rendering resumes as before. Figures 4a, 4b and 5a further demonstrate the versatility of our geometric library.

Dependency graph:

When a Geomatplot script runs, a dependency graph is built and then runs for the first time creating the initial plot. Then, the callbacks are called in order when an object is moved. Our performance optimization focuses on fast updates so the interactive plots feel as responsive as possible. Since the initialization performance is unimportant, we manually created many overloads of Geomatplot functions to maximize code readability and ease of use. Most of the code base deals with parsing the function arguments as many arguments can be optional such as label, color or line specification, line width or marker size, and initial position. When creating a dependent object, most functions accept a variable amount of Geomatplot entries, and either pass these to a user-defined callback function or depending on the type of these input geometries, the appropriate overload is called. For example, `Point([0 0])` creates a



(a) Geometric Newton-Raphson for closest point (b) Ordinary differential equation solution plot

Fig. 4: The slider controls the initial guess for the Newton-Raphson iteration to find the closest point on an arbitrary curve. On the right, MatLab's ode solver is called for the two orbiting celestial bodies in the co-rotating reference, and it calculates and plots the trajectory of a third massless projectile interactively.

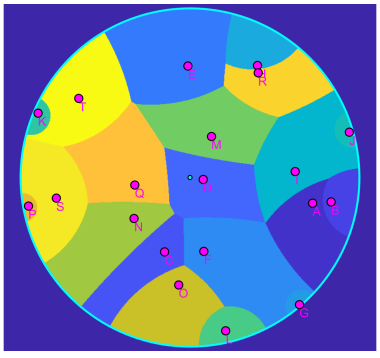
movable point, while `Point(A,B,@(a,b)0.5*(a+b))` creates the midpoint of `A` and `B`, same as calling `Midpoint(A,B)`. Another example is `Circle(O,r)`, `Circle(O,P)`, and `Circle(A,B,C)` which create a circle respectively with center `O` and radius `r`, center `O` and radius `Distance(O,P)`, and a circle through all three points `A`, `B`, `C`.

Performance:

Unfortunately, nested MatLab function calls introduce a large latency, so we had to make several optimizations. First, we created a flat dependency graph that lists the dependencies for each movable object and calls its callbacks in topological order. This also means that only what needs to be updated is changed, for example, on Fig. 3a, the background distance field is independent from the yellow point P and the update is much faster when P is moved. Second, many internal callbacks are inlined into each other wherever possible. Third, when the user writes expressions, such as `p0+t*v`, the results of these operations remain an unevaluated expression so subexpressions do not generate individual callbacks. The unevaluated expressions carry the symbolic expression with inputs and constants and generate the body of the callback when the expression is forced to evaluate.

Numerical considerations:

Somewhat surprisingly, drawing a line through two points requires more than just drawing a segment between two points extrapolated too far away from the initial two points. Such a line would not go through the two input points due to numerical error. For this reason, we draw a line from a short sequence of lines. Closest point, intersection, and distance calculations necessitated optimized algorithms for better performance and numerical accuracy. For example, our `Intersect` method relies on the general polyline with polyline intersections, except for circles with polylines and circles with circles for which we have implemented our numerically accurate and stable methods. Moreover, we improve responsiveness when drawing images by reducing the resolution while points are being dragged.



(a) Pseudo-hyperbolic Voronoi diagram on the unit disc

Examples	Figure	#lines	moving t.	stopping t.
Quadratic Bézier	Fig. 1b	15	2.62 ms	2.83 ms
Euler line of triangle	Fig. 2a	26	9.49 ms	9.24 ms
Sphere trace algorithm	Fig. 2b	12	6.59 ms	6.56 ms
$\mathbb{R}^2 \to \mathbb{R}$ image	Fig. 3a	14	4.52 ms	30.79 ms
Triangulation	Fig. 3b	16	3.75 ms	3.32 ms
Newton-Raphson	Fig. 4a	18	6.89 ms	6.87 ms
Ordinary diff. eq.	Fig. 4b	18	15.88 ms	26.03 ms
Hyperbolic Voronoi	Fig. 5a	21	20.59 ms	202.51 ms

(b) Code length and interactive update times for the presented examples. For real-time performance, image rendering lowers resolution when dragging points resulting in higher overhead (underlined) when releasing a point.

Fig. 5: Our last example on the left is a Voronoi diagram of the magenta points on the unit disc with the pseudo-hyperbolic metric $d(z,w) = \left|\frac{z-w}{1-z\overline{w}}\right|$ where $z, w \in \mathbb{C}$. The table on the right summarizes the number of lines – including comments and empty lines – and the performance of each example code.

Conclusions:

This paper demonstrated the effectiveness of Geomatplot with a variety of examples from different fields summarized in Table 5b. In general, our geometric library enables the quick creation of interactive plots with just a few lines of code. Thus, Geomatplot offers a programmable alternative to Geogebra with decent performance and offers a playground for research projects.

Future work:

The most significant bottleneck that causes noticeable latency when interacting with Geomatplot figures is that of the MatLab plotting library. Even though our implementation is more responsive than how Geogebra feels, and we directly update the plot data and use low-level MatLab plot commands, the rendering of such plots often lags behind our expectations. For this reason, we will be replacing MatLab's low-level drawing engine which will be more efficient due to OpenGL shaders, batch rendering, and viewport culling algorithms.

References:

[1] Hohenwarter, Judith; Hohenwarter, Markus: Introduction to GeoGebra. www.geogebra.org, https://tmtw.pbworks.com/f/geogebra-intro-en.pdf, 2008

[2] McMahon, David: MATLAB demystified. New York: McGraw-Hill, 2007.

[3] Quilez, Inigo; Jeremias Pol: Shadertoy BETA. https://www.shadertoy.com/about, 2013