

# A Semi-automatic Approach for Determining the Projection of Small Scale Maps Based on the Shape of Graticule Lines

Ádám Barancsuk

**Abstract** Knowing a map's projection is of essential importance, particularly when using them as a source for creating derivative works or dealing with them in a GIS environment. However, (especially on older maps), projection information is often absent or partially present. Our objective is to develop a semi-automated approach for determining the projection of a small-scale map, based on the shape and secondary properties of its graticule lines—as outlined in the hierarchy published by Érdi-Krausz (*Studia Cartologica* 1:194–270, 1958). To this end, a web-based tool is created, explicitly designed to be usable by a non-professional audience. Drawing tools are provided for manually tracing graticule lines on pre-uploaded raster maps. Given the approximate traces, we employ a number of algorithms to determine the shape and secondary properties (e.g. equidistance, concentricity, angles of intersection etc.) of graticule lines. Having computed these properties, one can fit the projection into Érdi-Krausz's system.

**Keywords** Projection analysis · Cartometry · Curve fitting

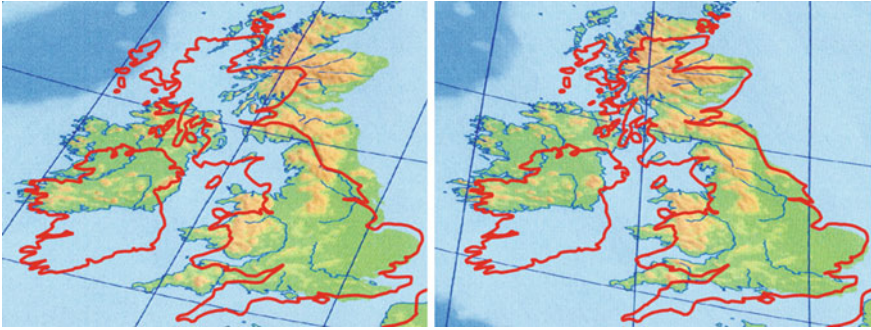
## 1 Introduction

Mix-and-matching already existing source maps and combining them to create derivative works is everyday practice in contemporary cartography. This was not always possible: historically, differences between projections among source works often posed major obstacles to such practices. The situation has improved substantially with the advent of GIS and computer-aided map design, that equipped cartographers with tools that now allow them to freely transform source material (both vector and raster datasets) into their projection of choice.

---

Á. Barancsuk (✉)

Department of Cartography and Geoinformatics, Faculty of Informatics,  
Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest 1117, Hungary  
e-mail: brnck@map.elte.hu; adam.barancsuk@gmail.com



**Fig. 1** Suboptimal results of georeferencing using different (global vs. local) interpolation techniques (*left* first order polynomials, *right* thin plate spline rubbersheeting)

How does one integrate datasets with different projections into one's cartographic workflow? This problem is dealt with in a number of different ways in cartographic practice. The most common solution involves ignoring the question of projections and using ground control points (with known coordinates in both a geographic and the dataset's own coordinate system) to associate the dataset with its location in geographic space. This method is called *georeferencing* and is very broadly used.

Georeferencing works by transforming points of the original dataset to new positions, while relying on the position of the ground control points as aid. Thus, georeferencing always involves a form of interpolation (either global or local), the precision of which is heavily dependent on the number and position of ground control points placed by the user. This method, if used improperly, can introduce gross transformation errors (see Fig. 1) that can render badly georeferenced datasets unsuitable for further use as described by Havlicek and Cajthaml (2015).

Another way to deal with the problem is taking the question of projections into account. This method assumes knowledge of both the *original* and the *target* projection to transform the dataset from/into. Compared to ground control point-based georeferencing which relies on points with known geographic *and* projected coordinates, this provides us with a *function* in a mathematical sense which, when applied to the original coordinates of each point, computes its coordinates in the target coordinate system. Such a method thus avoids interpolation and the errors associated with it, often leading to better precision.

However, information on the *original* projection of source works is often absent or only partially present. This is particularly true when using relatively old, printed maps as a source: while digital cartographic data are usually accompanied by metadata including a projection definition, paper-based maps seldom provide such information with sufficient level of detail. (Note, that for the purpose of using maps in a GIS environment, not only the type of projection but also its exact parameters should be known to the user.)

To overcome the problem of determining unknown cartographic projections, this study proposes a semi-automatic approach for making educated guesses regarding the projection of a small-scale map. Our guess does not only serve as an aid for more precise georeferencing but can also provide metadata that might be useful when digitizing and cataloguing cartographic documents in a library.

The method we describe primarily builds on the work of György Érdi-Krausz who proposed a system for determining a small-scale map's projection by examining its graticule (geographic grid). This system was first outlined in a study concerned with the analysis of cartographic projections (Érdi-Krausz 1958) then later refined by Györfy (2012). Érdi-Krausz's study describes an algorithm for determining an unknown projection which is modeled as a *decision tree*. The algorithm starts at the tree's root node, ascends the tree by repeatedly branching based on different properties (such as the shape, position, intersections and spacing) of the map's graticules lines, then arrives at one of the leaves containing a specific type of projection.

While Érdi-Krausz's study also describes projection-specific methods that allow the user to compute projection parameters, our research in its current state has a narrower but more practical focus: First, we utilize applied mathematical methods to try and implement Érdi-Krausz's decision tree algorithm on computer. Second, we build a web-based application that helps people determining the projection of a small scale map, who otherwise lack deeper knowledge of cartographic projections.

## 2 Related Work

Many online tools are available for georeferencing maps, but none of them deals with the question of projections. This may be mainly due to the fact that the problem of determining the projection of maps, then using the resulting information for the purposes of georeferencing has not been studied in great depth historically. Such analysis would have required abundant computing resources that were unavailable even a couple of decades ago. In the recent years however, several approaches were developed that allow us to make educated guesses about unknown cartographic projections.

According to the author's knowledge, there are three methods that actually solve the problem of determining unknown projections of maps. The first one is an application developed by Snyder (1985). The main focus of Snyder's work is to allow data transfer between two maps in different projections. His algorithm is similar to the one described in this paper in that it also uses a decision tree mechanism to determine the projection type. The list of recognized projections includes the most commonly used ones from every major class (azimuthal, cylindrical, pseudocylindrical and conic) and also includes a second-order polynomial approximation of the projection as a last resort. Unlike our method which works on graticule lines, Snyder's program only takes nine points as its input, which should reside along three meridians and three parallels. Given this information, the

program performs a number of measurements on the graticule: it checks whether graticule lines are straight or curved, the spacing of meridians and the concentricity of parallels. Because of it only relies on nine known points, Snyder's algorithm provides a more rudimentary view on the form of the graticule (e.g. it cannot differentiate between different curves). Compared to our approach, however, his program also provides the exact parameters for the projection.

The other two methods are MapAnalyst (described thoroughly by Jenny and Lorenz 2011), and Detectproj, which is developed by Bayer (2014) and seems to provide the most comprehensive solution to the problem to date.

These latter methods, while they differ in important aspects of their implementation, view the problem in a roughly similar way:

- Consider a predefined list of projections and a set of control points (and possibly other, higher dimensional features such as lines and polygons) with coordinates both known in geographic space and the unknown map's own coordinate system.
- Iterate through the list of possible projections. For each projection:
  - Iterate through a range of different projection parameters or apply numerical optimization to traverse the parameter-space towards the best fit:

Transform the set of control points (and possibly other features) from geographic space into the current projection and parameter set.

Assess the difference between the set of freshly-transformed features and the set of features on the unknown map.

Compute some measure that quantifies the goodness of fit (transformation parameters) between the two sets.

- Having found the projection with the highest goodness of fit, report that as the most likely projection for the map.

Our method takes a different approach:

- Consider a set of graticule lines on the unknown map, each defined by an ordered set of its points.
- Traverse the decision tree by on-demand computation of the following properties of the graticule:
  - Best fit **curve type** for the given graticule line (in its current state, our algorithm can distinguish between straight lines, circles, generic ellipses, parabolas and hyperbolas);
  - Position, spacing and angle of intersection at **intersection points**;
  - **Concentricity of the curves** representing graticule lines;
  - **Representation of poles** (lines or points);
  - **Closedness of curves**.

These differences result in a number of advantages and disadvantages when compared to the former two algorithms:

- We found the decision tree traversal method to be faster (5–10 s for the whole graticule) than Detectproj iterating through a set of projections (couple of minutes). This may also be due to the fact that our method of choice for numeric optimization takes derivatives into account which results in faster convergence.
- Our method in its current form is unable to take different aspects and projection parameters into account. While this lets us focus on a subset of projections that are of actual cartographic importance (without having to deal with existent, but rarely used variants), it also severely narrows the feasibility of our method. This also means we cannot easily quantify the goodness of fit of our solution or offer multiple possible choices sorted by that measure.
- To extend our method with new projections, one has to fit the given projection into Érdi-Krausz's system. This is possible but far from straightforward. To be extended, the previously discussed methods only require the projection's mathematical formulation to be known.

### 3 Mathematical Foundations

As mentioned above, the primary goal of our research is to automate the computations needed to traverse the decision tree of Érdi-Krausz's system. As an input, the algorithm takes a set of line strings (ordered set of points) that represent a graticule line. In its current state, the application requires the user to manually trace the graticule lines on the previously uploaded raster map.

#### 3.1 *Homogeneous Coordinates*

Throughout the application we use *homogeneous coordinates* to represent points. Homogeneous coordinates are coordinate vectors used in projective geometry. One advantage of homogeneous coordinates is that they allow us not only to represent points of Euclidean spaces but also points at infinity. They are also widely relied upon in computer graphics as they allow formulas of affine (or more generally, projective) transformations to be represented as matrix operations. We use them because they allow us to formulate relationships between points and curves of our graticule more easily than it would be possible with Euclidean coordinates.

Given a point  $(x, y)$  on the Euclidean plane and for any  $z \in \mathbb{R} \setminus \{0\}$ , the homogeneous coordinate-vector of the point can be written as  $(xz, yz, z)$ .

### 3.2 Curve Fitting

Traces of graticule lines drawn by the user inherently has a small amount of error introduced by the limited accuracy of the original maps and also the tools provided for drawing. This means that no straight line, conic section or other mathematically representable quadratic curve can be fit to these points exactly.

Both curve fitting problems we consider in this section can be formulated as the solution of a system of  $N$  equations in the form of

$$f(x_i, \mathbf{p}) = 0$$

where  $x_i (i = 1 \dots N)$  are coordinate vectors of the points to be fit and  $\mathbf{p} \in \mathbb{R}^M$  is the unknown vector of parameters of the curve in question. Such a system is said to be *overdetermined* when  $N > M$ , meaning it consists of more equations than the number of unknowns. This is usually the case when fitting user-drawn traces of a couple dozen points to lines or conic sections defined by either 3 or 5 parameters.

The above mentioned two circumstances makes curve fitting an optimization problem. As such, we have to find a method that provides an approximate solution to our problem, characterized by some minimal measure of error.

A possible definition of error is a vector of corrections (or deviations)  $\mathbf{r} \in \mathbb{R}^N$  such that, when applied to the original system  $f(\mathbf{x}, \mathbf{p}) = 0$  it is transformed into  $f(\mathbf{x} + \mathbf{r}, \mathbf{p}) = 0$ , which now has an actual solution. Thus our goal is to find some vector  $\mathbf{p}$  for which the error measure  $\mathbf{r}$  is minimal. As is customary in statistics, we choose to optimize the square of errors which leads us to the optimization problem:

$$\min \sum_{i=1}^N \|r_i\|^2$$

This is the method of *least squares*, that is the most frequently used method in regression analysis for finding approximate solutions to overdetermined systems.

In our application we utilize a two-pass least-squares fitting algorithm to try and fit straight lines and conic sections to the points of traces drawn by the user. This algorithm is run on each graticule line separately. First, the algorithm tries to fit a straight line to the given set of points (see Sect. 3.2.1). If the error of this fit is lower than a predefined numeric tolerance, the fit is accepted and the line is considered to be straight. The second, conic fitting stage (see Sect. 3.2.2) is only performed if straight line fitting fails (the error is higher than the given tolerance). After conic fitting is performed, the goodness of the second fit is again compared to another numeric tolerance. If both stages fail (both errors are above the respective tolerances), the application considers the graticule line to be of an unknown curve type and excludes it from further processing (i.e. no secondary properties are computed on these lines). If either of the fitting passes succeed, the result is a parameter vector of the best fit straight line or conic section, that is associated to the graticule line.

The later stages of our program only use these parameter vectors to represent the graticule lines.

Theoretically, a single-pass fitting algorithm would also be possible, because a straight line is just a special (degenerate) case of a conic section. However, fitting degenerate conic sections proved to be numerically unstable in practice.

### 3.2.1 Line Fitting

On the Euclidean plane, a line may be formulated as  $y = ax + b$  (the so-called slope-intercept form). This form however does not allow us to represent vertical lines which are ubiquitous when dealing with map graticules. To overcome this limitation, we use the line equation from projective geometry,  $ax + by + c = 0$  which can also be represented with the three-element homogeneous coordinate-vector  $(a, b, c)$ . Note that in the two dimensional projective space, the same coordinate-triple may be used to represent either points or lines. This leads to the notion of *duality* in projective geometry.

Given the line  $l = (a, b, c)$  and the point  $p = (x, y, 1)$ , said point lies on the line when it satisfies the equation  $ax + by + c = 0$  of the line. This can also be written using homogeneous coordinates as  $pl = 0$ .

Our goal is to find a line that fits best to the points of the user-drawn trace of a graticule line. Given  $N$  trace points this yields a system  $A$  of  $N$  linear equations in the form mentioned above.

$$A = \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_N & y_N & 1 \end{pmatrix}$$

Given the above the problem of line fitting in projective two-space can be formulated as the equation  $Ax = 0$  where  $x \in \mathbb{R}^3$  represents the coordinate triple of the solution line.<sup>1</sup>

To obtain the solution we consider the singular value decomposition (SVD) of the matrix  $A$ . The singular value decomposition is a factorization in the form

$$A = U\Sigma V^T$$

which can be proven to exist for all real matrices  $R \in \mathbb{R}^{N \times M}$  (Bast 2004). The product on the right hand side consists of the unitary matrices  $U \in \mathbb{R}^{N \times N}$  and  $V \in \mathbb{R}^{M \times M}$  (called the collection of left and right singular vectors), and the  $\Sigma \in \mathbb{R}^{M \times N}$  rectangular diagonal matrix, the elements of which are all zeroes except for

---

<sup>1</sup>Every homogeneous linear system  $A$  is satisfiable by the trivial solution  $\forall x \in x = 0$ . They have non-trivial solutions only if  $\text{rank}(A) < N$ .

the values  $\sigma_1 > \sigma_2 > \dots > \sigma_n > 0$  in its main diagonal. The vector composed from the biggest elements of the right-singular vectors

$$\mathbf{x} = \left\{ V_{m,1}^T, V_{m,2}^T, \dots, V_{m,n}^T \right\}$$

can be proven (Tomasi 2013) to be the least-squares solution for the system  $A$ . In our case,  $N = 3$ , thus the solution is a vector  $\mathbf{x} \in \mathbb{R}^3$ , which is the parameter vector of the best-fit line in projective space.

### 3.2.2 Conic Section Fitting

Given

- A number of points in projective 2-space  $P = \{p_i\}$  where  $p_i = (x_i, y_i, 0)$ ,
- A group of conic sections  $C(\mathbf{a})$  where  $\mathbf{a} \in \mathbb{R}^5$  is the vector of parameters for the conic section,
- And some distance measure  $\delta(C(\mathbf{a}), x_i)$ , describing the distance between  $\mathbf{x}$  and  $C(\mathbf{a})$

our aim is to determine the vector of parameters  $\mathbf{a}_{min}$  for which the least-squares measure of error

$$\epsilon^2(\mathbf{a}) = \sum_{i=1}^n \delta(C(\mathbf{a}), x_i)$$

is minimal.

This problem is well-researched in the field of computer vision and graphics. For an overview of methods for solving it, see Fitzgibbon and Fisher (1995). Most of the techniques consider only a certain type of conic sections (e.g. ellipses, parabolas or hyperbolas), but our use case requires a method that is agnostic on the type of the conic section.

Conic fitting methods can also be characterized based on their choice of distance measure. The most easily computable form of distance is *algebraic distance*. A conic section can be defined as the quadratic form

$$C(\mathbf{a}) = a_1x^2 + a_2xy + a_3y^2 + a_4x + a_5y + a_6 = 0.$$

Algebraic distance then can be formulated as

$$\delta(c(\mathbf{a}), \mathbf{x}) = \mathbf{x}^T A_Q \mathbf{x}$$



where

$$A_Q = \begin{pmatrix} a_1 & \frac{a_2}{2} & \frac{a_4}{2} \\ \frac{a_2}{2} & a_3 & \frac{a_5}{2} \\ \frac{a_4}{2} & \frac{a_5}{2} & \frac{a_6}{2} \end{pmatrix}$$

The problem with algebraic distance is that it often results in geometrically (visually) inaccurate fits. Therefore, we choose to optimize *orthogonal distance* which is believed to be the most natural choice (Ahn 2004).

Computation of orthogonal distances between points and a general conic is possible, but can be numerically instable. This can either be overcome by using properties that are specific to a type of conic such as rotation, position of the center etc., or by using the method described in Wijewickrema et al. (2010).

To be able to compute the orthogonal distance between a point  $\mathbf{b}$  and the conic  $C$ , the authors choose to solve the system

$$\begin{aligned} \mathbf{x}^T C \mathbf{x} &= 0 \\ \mathbf{x}^T D \mathbf{x} &= 0 \end{aligned}$$

where

$$D = c_1 a_1^T - c_2 a_1^T$$

is another conic section that represents the relationship between  $C$  and  $\mathbf{b}$ . Solving the system yields two intersection points. We consider the one closer to  $\mathbf{b}(\mathbf{x})$  to be the orthogonal point of  $\mathbf{b}$  on  $C$  (Fig. 2). The solution of the system leads to a quartic formula. To evade numeric instabilities that emerge when solving the formula, the authors instead choose a method that relies on finding a degenerate conic:

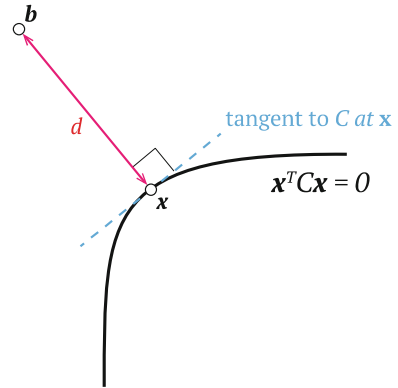
$$C_d = \mathbf{l} \mathbf{m}^T + \mathbf{l}^T \mathbf{m}$$

then decomposing it into two straight lines  $\mathbf{l}$  and  $\mathbf{m}$  which allow us to compute the intersection points between either line and the original conic  $C$ . This results in two possible intersection points. As mentioned above, we choose the point closer to  $\mathbf{b}$  and consider the distance between  $\mathbf{b}$  and that point to be the orthogonal distance between  $\mathbf{b}$  and  $C$ . Having computed the orthogonal distance  $\delta(C(\mathbf{a}), \mathbf{b})$ , we have to solve the optimization problem

$$\min \sum_{i=1}^n \delta(C(\mathbf{a}), \mathbf{b}_i)$$

for which purpose we choose the Levenberg–Marquardt method as recommended by Wijewickrema et al. (2010). This iterative method requires the calculation of the Jacobian matrix for each step which makes the process robust but more

**Fig. 2** Orthogonal distance between a conic section and a point



computationally expensive compared to other methods. Reliance on the Jacobian also makes the method insensitive to the choice of initial parameter vector, therefore no prior fitting is required.

### 3.3 Curve Intersections

Having computed the best fit curve for all graticule lines, we also have to obtain the position and angle of their points of intersection. Based on this information we can differentiate between certain members of projection families. (As an example, one might consider either (pseudo)cylindrical or azimuthal projections. In both cases, the change in spacing between lines of latitudes towards the map contour is indicative of distortion conditions.)

#### 3.3.1 Intersection Points

Currently we only consider intersection points between one line of latitude and one line of longitude (but not two lines of latitude or longitude). The remaining two cases always fall into either one of the following categories:

1. The **intersection of two straight lines** defined by their projective coordinates  $(a_1, b_1, c_1)$  and  $(a_2, b_2, c_2)$  can be found by computing

$$(a_1, b_1, c_1)^T \times (a_2, b_2, c_2)^T$$

2. Assuming that both input triples describe lines that are not in the infinity, the Euclidean coordinates of the intersection point can be inferred from the resulting triple of projective coordinates.

3. The **intersection of a straight line and a conic section** can be computed by first computing the antisymmetric matrix of the line **l**:

$$L = \begin{pmatrix} 0 & -l_3 & l_2 \\ l_3 & 0 & -l_1 \\ -l_2 & l_1 & 0 \end{pmatrix}.$$

Given the above

$$D = L^T C L$$

is a degenerate conic, which represents the intersection points between  $C$  and **l**. One can decompose  $D$  to obtain these intersection points.

### 3.3.2 Intersection Angles

The angle between two homogeneous lines is given by

$$\Theta = \cos^{-1}(l_1 m_1 + l_2 m_2)$$

whereas to compute the angle of intersection between a straight line and a conic section, one first has to calculate a tangent line to the conic at the intersection point. This is given by

$$l = C p^T.$$

Having computed the tangent line, one might obtain the intersection point by intersecting the tangent with the other straight line.

### 3.3.3 Intersection Spacing

To distinguish between projections with different distortion conditions, the spacing between intersection points also has to be computed. In its current state, the application only considers distances between intersection points lying on a straight line of graticule. Positions of intersections along this line may be characterized by the homogeneous vector-parametric equation of that line:

$$P = P_0 + t r$$

where  $P$  is the graticule line,  $P_0$  is an arbitrary point along that line,  $r$  is the directional vector of the line and  $t \in \mathbb{R}$  is a scalar indicating the position of  $P$  along the line relative to  $P_0$ .

Given our lines with the homogeneous triple  $(a, b, c)$  and also taking vertical lines into account:

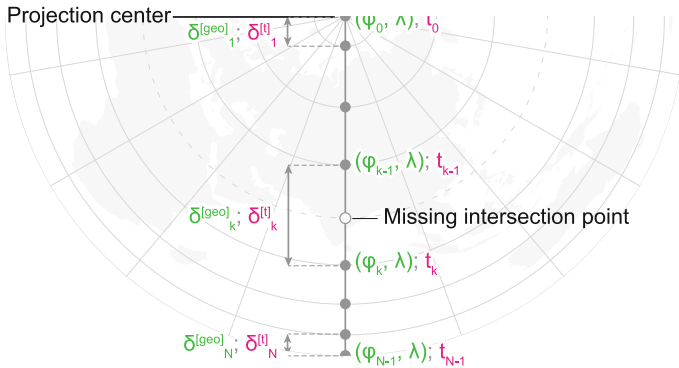


Fig. 3 Computation of the intersection points

$$P_0 = \begin{cases} \left(-\frac{c}{a}, 0\right)^T & \text{if } a \neq 0 \text{ and } b = 0 \\ \left(-\frac{a}{b}, -\frac{c}{b}\right) & \text{otherwise} \end{cases},$$

the directional vector is:

$$r = \begin{cases} (1, 0)^T & \text{if } a = 0 \text{ and } b \neq 0 \\ (0, 1)^T & \text{if } a \neq 0 \text{ and } b = 0 \\ \left(-\frac{a}{b}, -\frac{c}{b}\right) & \text{otherwise} \end{cases}.$$

and the relative position of the intersection point along the line can be obtained by solving the system defined by the vector parametric equations for  $t$ .<sup>2</sup>

To be able to assess how the spacing of intersection points change when moving along a straight graticule line from the projection center towards the map contour, we have to compare changes in geographic coordinates ( $\delta^{[geo]}$ ) with planar distances (that is, changes in intersection point positions  $\delta^{[t]}$ ).

One might expect that comparison of values of the parameter  $t$  for different intersection points would also be sufficient, but that would rely on the implicit assumption that graticule lines are geographically equally spaced (e.g. every  $10^\circ$ ). This is not the case when graticule lines are more densely spaced in an area of the map than in another. The assumption might also be incorrect in a number of other cases. Such an example might be when the curve fitting algorithm fails on a user-drawn trace for some reason, thus intersection points along that curve cannot be computed (Fig. 3).

<sup>2</sup>In the application's current implementation, the solution is obtained by using the `\` (backslash) operator in Julia which is based on a form of QR-factorization (see <http://docs.julialang.org/en/release-0.4/stdlib/linalg/#Base>).

For obtaining the planar distances we compute:

$$\delta^{[geo]} = \{ \kappa_j - \kappa_i | \forall i, j \in [1, N] \text{ and } j = i + 1 \}$$

as well as a vector of changes of the geographic coordinate:

$$\delta^{[l]} = \{ t_j - t_i | \forall i, j \in [1, N] \text{ and } j = i + 1 \}$$

where  $N$  is the number of intersection points and  $\kappa$  is the respective geographical coordinate ( $\lambda$  when measuring spacing along a line of latitude and  $\varphi$  when measuring spacing along a line of longitude). To compare the two, we look at the geographic differences normalized by planar distances:

$$\delta = \left\{ \frac{\delta_i^{[geo]^2}}{\delta_i^{[l]^2}} - i \in [1, N] \right\}.$$

This yields distances that are independent from the geographical spacing of graticule lines on the map. To assess the equidistancy of the intersection points, we compute the coefficient of variation of  $\delta$ :

$$c_v = \frac{\sigma}{\mu}, \text{ where } \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N \delta_i - \mu^2} \text{ and } \mu = \frac{1}{N} \sum_{i=1}^N \delta_i.$$

The coefficient of variation measures the variation of the distribution of the normalized distances, irrespective of the distances' actual values. The spacing of intersection points is considered uniform if  $c_v$  is lower than a predefined numeric tolerance.

### 3.4 Concentricity

To assess whether certain curves of graticule are concentric, one should compute the position of their center points. This helps us differentiate between pseudoconic projections, where curves of latitude are concentric and polyconic projections, where they are not.

Computation of the center point is only done in case of ellipses and hyperbolas. The center point of a generic conic section lies where the gradient of the quadratic form of the conic (see Sect. 3.2.2) vanishes:

$$\nabla C = \left( \frac{\partial C}{\partial x}, \frac{\partial C}{\partial y} \right) = (0, 0).$$

By substituting the partial derivatives into the quadratic form  $C$ , we obtain

$$\begin{cases} \frac{a_4}{2} + a_1x + \frac{a_2}{2}y = 0 \\ \frac{a_5}{2} + \frac{a_2}{2}x + a_3y = 0 \end{cases}$$

If the determinant of this system is non-vanishing (i.e. the conic is either an ellipse or a hyperbola), the Euclidean coordinates of the center is given by  $(x, y)$  (Wylie 2011). To assess whether conic sections representing graticule lines share the same center, the standard deviation of centers is computed. A numeric tolerance for this value is defined, which, when not exceeded by the standard deviation, indicates the concentricity of graticule lines.

### 3.5 Representation of Poles

There are a number of ways to determine how poles are represented on the map. For the sake of simplicity, we expect the user to also trace poles when they are represented by lines on the map. In case poles are mapped to points, another drawing tool is provided on the user interface that allow the user to trace these points. When curves of latitude with a coordinate value of  $\phi = \pm 90^\circ$  are present, poles are considered to be lines. When points are present, poles are considered to be points. When none or both of the above is true, representation of poles is assumed to be undefined and the property is ignored when traversing the decision tree.

### 3.6 Closedness

Again, for the sake of simplicity, closedness of curves is computed as the norm of differences between the positions of the first and last point of the corresponding trace along the  $x$  and  $y$  axis of the viewport coordinate system on which tracing was done.

### 3.7 Decision Tree

After computing all necessary properties, we can traverse the decision tree to obtain the type of projection (Fig. 4). In its current form the decision tree is implemented as a series of nested conditional branches of program code. This is hard to maintain and is planned to be reimplemented such that projection definitions can be specified declaratively.

When traversing the tree, decisions are often made based on properties of a *group of objects* at once (e.g. “Which type of curve fits best to *all* of our latitude lines?”) As the inherent error of user-drawn traces may affect the precision of our algorithms (as seen in Sect. 5.1), they may provide bogus results for certain

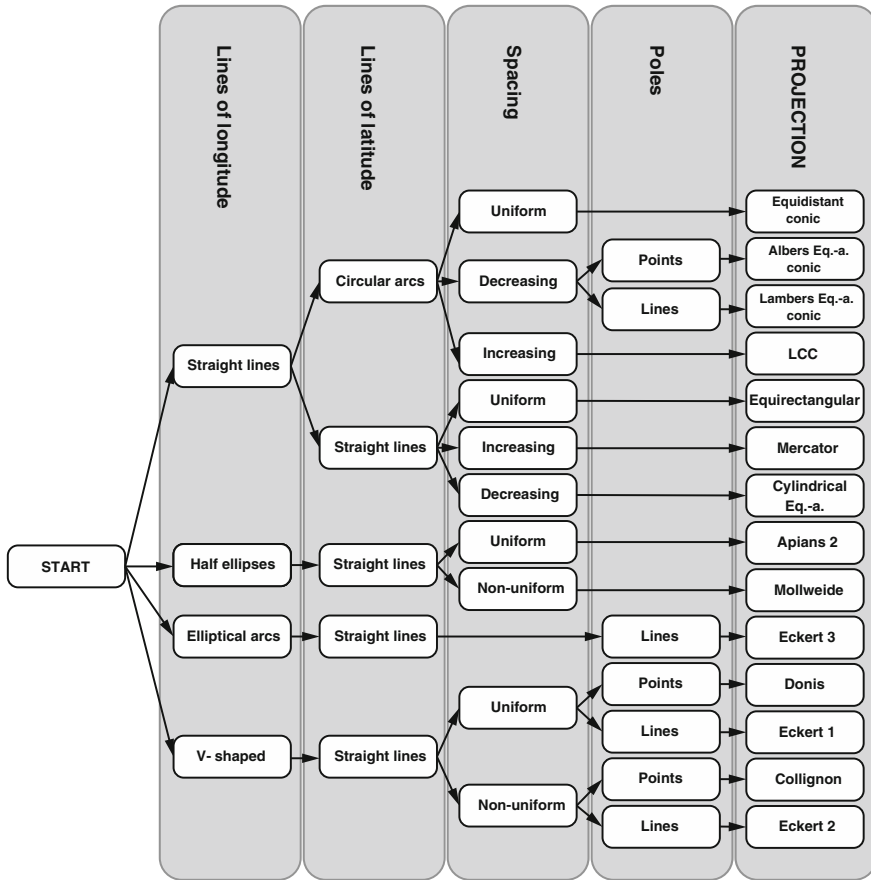


Fig. 4 Excerpt from the decision tree used for categorizing projections (some levels are omitted for clarity)

graticule lines. To compensate for this situation, when inferring a property of a group of objects we calculate said property on each object, then assume the most frequent value in the dataset to be the answer. This allows us to proceed even when one of our algorithms fail for a graticule line.

## 4 Application

To test the algorithms outlined in Sect. 3, we developed a web-based application. The two-tier architecture (Fig. 5) of the application allow us to separate business logic from user interface concerns and forces us to explicitly specify means of data transfer between these two layers.

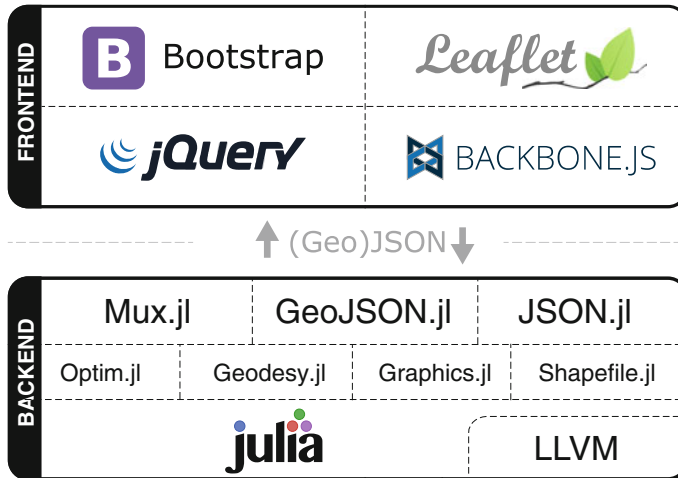


Fig. 5 Application architecture

#### 4.1 Backend

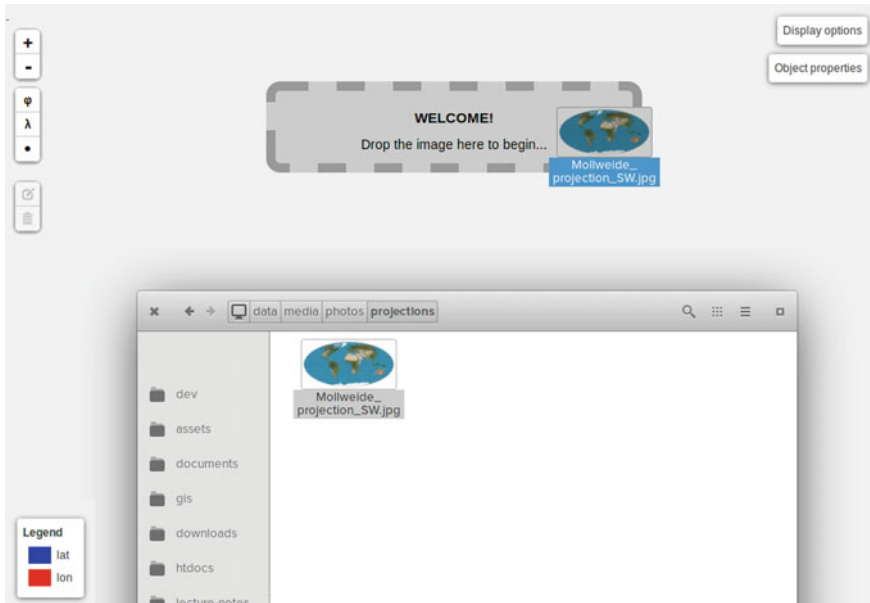
The application's backend where the previously discussed algorithms run was developed using the Julia programming language. Julia<sup>3</sup> is a high-level, high-performance dynamic programming language created for the purposes of technical computing. Its user-facing characteristics roughly resemble those of other popular technical languages as Matlab and R, however Julia also provides a powerful just-in-time compiler based on the LLVM compiler architecture that makes it possible to write highly performant applications. With its Git-based package management system, Julia was easily extendable for our purposes. For example the packages `JSON.jl` and `GeoJSON.jl` handles data (de)serialization between the server and the client, `Optim.jl` contains an implementation of the Levenberg–Marquardt method we use for curve fitting, while `Mux.jl` provides an HTTP server implementation for communicating with the client.

#### 4.2 User Interface

The user interface is based on widely used contemporary web technologies. We chose to implement a web-based interface for our application because we believe the web is the most widely accessible platform that is available independent of the operating system and other architectural differences between devices.

<sup>3</sup><http://julialang.org>.





**Fig. 6** Uploading a raster map for processing on the user interface of the application. Tools for tracing graticule lines and poles can be seen in the *upper left* corner

The application consists of a map view into which raster maps can be uploaded by the user (Fig. 6). An uploaded map, its graticule lines and the result of backend algorithms together form a *project* that can be saved, then restored at a later time. After uploading, the user is advised to manually trace the graticule of the map using a number of tools that are provided. The interface resembles that of widely used web based mapping applications to maintain a familiar feel.

After tracing each graticule line the user should input the value of the geographic coordinate associated with the freshly drawn curve. When all lines are traced, backend processing can be started. At this point, all traces are uploaded to the server where assessment of graticule properties and traversal of the decision tree to determine the projection is automatically done.

When finished, results of the backend processing are presented to the user in a table. The guessed projection, processing times as well as resource usage are also reported.

The user interface also features a sidebar where common operations such as saving/loading projects and setting application parameters (such as numeric tolerances) can be invoked. Also provided is a panel for modifying how graticules are displayed. The user can choose between displaying graticule lines based on their coordinate type (latitude/longitude), curve type (straight line, ellipse, parabola etc.), equidistancy of their intersection points or closedness.

## 5 Testing and Results

### 5.1 Synthetic Testing

We first tested the application by programmatically generating graticules for widely used projections.

The algorithm for graticule generation takes the following parameters:

- **Projection type** ( $p$ )—For testing purposes we included a number of projections in the application by implementing their projection equations. We tested the algorithm with projections from each major projection class (azimuthal, cylindrical and conic).
- **Graticule interval** ( $n$ )—Controls the density of the generated graticule. By gradually decreasing this value, processing time increases with the number of generated graticule lines. Having a lot of graticule lines also helps in testing other algorithms such as intersection point finding.
- **Scale** ( $s$ )—All projections implemented for testing purposes are computed as if they were on a unit sphere. For display purposes and to avoid numerical problems, an up-scaling factor had to be defined.
- **Noise** ( $\sigma$ )—As mentioned in Sect. 3.2 user-drawn traces are inherently noisy. In order to simulate this, we added a configurable pseudorandom noise modifying the position of the graticule lines' points. For the sake of simplicity, we assumed the noise to be normally distributed: ( $\mu = 0, \sigma$ ). This allowed us to define the amount of noise by specifying its variance  $\sigma$ .

Given these parameters we could generate our synthetic graticules by computing

$$P_{\Lambda} = \left\{ \left( \left[ \begin{array}{c} x_p(\text{arc } \Lambda, \text{arc } \phi) \\ y_p(\text{arc } \Lambda, \text{arc } \phi) \end{array} \right] + \left[ \begin{array}{c} \epsilon_1 \\ \epsilon_2 \end{array} \right] \right) * s \mid \phi \in \{ \phi_{\min} \leq x \leq \phi_{\max} \mid x \bmod n = 0 \} \text{ and } \epsilon_{1,2} \sim \mathcal{N}(\mu = 0, \sigma) \right\}$$

and

$$P_{\Phi} = \left\{ \left( \left[ \begin{array}{c} x_p(\text{arc } \lambda, \text{arc } \Phi) \\ y_p(\text{arc } \lambda, \text{arc } \Phi) \end{array} \right] + \left[ \begin{array}{c} \epsilon_1 \\ \epsilon_2 \end{array} \right] \right) * s \mid \lambda \in \{ \lambda_{\min} \leq x \leq \lambda_{\max} \mid x \bmod n = 0 \} \text{ and } \epsilon_{1,2} \sim \mathcal{N}(\mu = 0, \sigma) \right\}$$

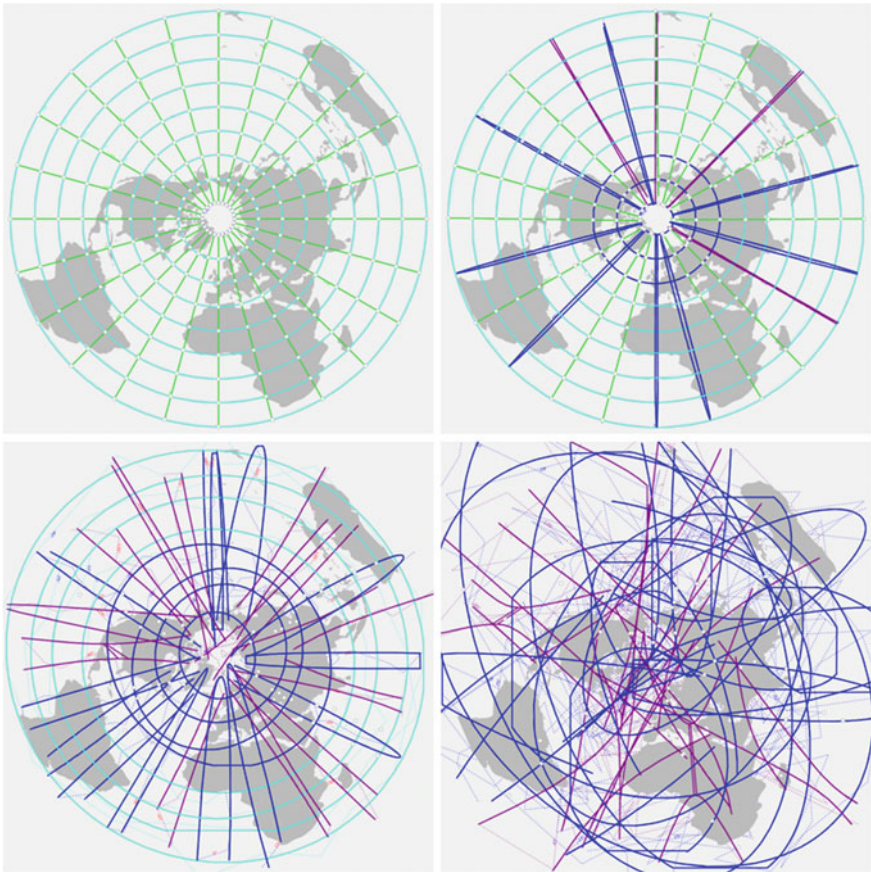
for every  $\Lambda$  and  $\Phi$  where they are an exact multiple of  $n$ . For lines of longitude,  $P_{\Lambda}$  are the set of points generated for the line where the longitude equals  $\Lambda$  degrees.  $x_p$  and  $y_p$  denote the projection equations for the chosen projection  $p$ ,  $\epsilon_{1,2}$  is the pseudo-random noise generated from the normal distribution  $\mathcal{N}(\mu = 0, \sigma)$ , and  $s$  is the up-scaling factor defined earlier. Points in  $P_{\Lambda}$  are generated such that  $\Lambda$  stays constant, while  $\phi$  is a parameter moving from  $\lambda_{\min}$  to  $\lambda_{\max}$  sampled at the chosen interval defined by  $n$ .

The same is true for lines of latitude ( $P_{\Phi}$ ) when exchanging  $\Lambda$  with  $\Phi$  and  $\lambda$  with  $\phi$  respectively. For certain projections (e.g. the Mercator projection), graticule

boundaries ( $\phi_{min}$ ,  $\phi_{max}$ ,  $\lambda_{min}$ ,  $\lambda_{max}$ ) should be limited so as to avoid generating graticules lines which lie at infinity.

Testing our algorithms with these synthetically generated graticules proved our implementation to be quite robust. Increasing the graticule density did not seem to have a significant effect on processing time. On the other side, we saw that increasing the amount of noise resulted in significantly worse accuracy during curve fitting. As curve fitting is the first step in the row of our algorithms, erroneously recognizing a curve means all subsequent steps would fail and projection recognition might either become impossible or—worse—give an incorrect result.

By comparing synthetically generated test data with graticule lines traced by humans we empirically came to the conclusion that errors introduced by manual tracing fall into an order of magnitude best approximated by setting the synthetic



**Fig. 7** The effect of pseudo-random noise with differing variance on the accuracy of curve fitting (azimuthal equidistant projection)

noise variance to  $\sigma = 10^{-2}$ . As it can be seen on Fig. 7, errors of this magnitude are relatively well handled by our algorithm, thus it remains suitable for practical use.

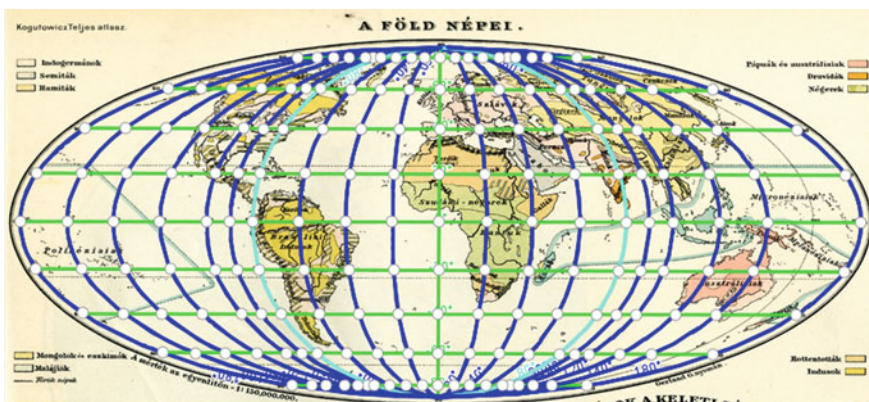
The success of curve fitting and a number of other steps are also dependent on user definable numeric tolerances. In its current state, the application relies on the user to set these values before computations, however empirically established defaults are also provided in the configuration file which should work well most of the time.

## 5.2 Testing on Real Maps

For the purpose of testing the algorithm on real maps we used maps from the geographic atlas *Teljes földrajzi atlasz* (Complete Geographic Atlas) a collection of maps from the beginning of the 20th century, authored by Hungarian cartographer and publisher Manó Kogutowicz, that—besides geographic themes—also includes thematic maps in a variety of cylindrical and pseudocylindrical projections (Kogutowicz 1902).

Due to the rather limited scope of these experiments, the results should only be considered as a proof of our concept, not a comprehensive suit of tests. Even so, trying out our algorithms on real maps already yielded some interesting results. When applied to one of the maps in *Teljes atlasz*, the application correctly identified its projection as being Mollweide. However, longitudes of  $\pm 80^\circ$  were identified as circles instead of  $\pm 90^\circ$  as should be the case for a map in the Mollweide projection (Fig. 8).

This calls attention to the problem of the inaccuracy of hand drawn maps. We expect to encounter this problem frequently as we deal with historical cartographic documents that might not have a solid geodetic and geometric basis. This is a problem that needs to be dealt with in a future version of the application.



**Fig. 8** An interesting result: longitudes of  $\lambda = \pm 80^\circ$  (instead of  $\pm 90^\circ$ ) are identified as circles on a map in the Mollweide projection (Green lines; blue ellipses; cyan circles)

### 5.3 List of Recognized Projections

Having implemented the algorithms presented in the previous sections, our application can distinguish between the following types of projections on small scale maps:

<b>Cylindrical</b> 1. Equirectangular 2. Mercator 3. Lambert Cylindrical Equal-area	<b>Conic</b> 16. Equidistant Conic 17. Lambert Equal-area Conic 18. Albers Equal-area Conic 19. Lambert Conformal Conic
<b>Pseudocylindrical</b> 6. Apian's 1/2 (Ortelius) 7. Sinusoidal 8. Mollweide 9. van der Grinten's 3 10. Eckert 3–6 11. Kavrayskiy 1–2, 6–7	<b>Azimuthal</b> 20. Lambert Azimuthal Equal-area 21. Azimuthal Equidistant 22. Stereographic 23. Gnomonic (polar/normal aspect)
	<b>Other</b> 24. Hammer 25. Aitoff 26. Pseudoconic

## 6 Conclusion

In our study we examined the possibilities to implement Érdi-Krausz's decision tree system for determining the projection of small scale maps on a computer. We presented the mathematical foundations for the work and gave an overview of the accompanying web-based application that has been developed to aid non-professional users in the process of determining unknown projections on raster maps. We tested our algorithms both with synthetically generated graticules and on real maps. Both set of tests found the algorithm to be generally useable in practice, however a number of problems were revealed which might serve as a basis for further research.

## References

- Ahn SJ (2004) Least squares orthogonal distance fitting of curves and surfaces in space. Doctoral thesis, University of Stuttgart
- Bast H (2004) An existence proof for the singular value decomposition. <https://www.cs.duke.edu/courses/fall13/compsci527/notes/svd.pdf>. Accessed 16 Mar 2016

- Bayer T (2014) Estimation of an unknown cartographic projection and its parameters from the map. *Geoinformatica* 18:621–669. doi:10.1007/978-3-642-11840-1\_19
- Érdi-Krausz G (1958) Vetületanalízis. (Analysis of Cartographic Projections). *Térképtudományi tanulmányok (Studia Cartologica)* 1:194–270
- Györffy J (2012) Térképészet és Geoinformatika II. Térképvetületek. (Cartography and Geoinformatics 2. Cartographic Projections)
- Fitzgibbon A, Fisher R (1995) A buyer's guide to conic fitting. In: *Proceedings of the British Machine Vision conference*, pp 513–522
- Havlicek J, Cajthaml J (2015) Influence of replacement used reference coordinate system for georeferencing of the old map of Europe. *Int J Environ Chem Ecol Geol Geophys Eng* 9: 749–753
- Jenny B, Lorenz H (2011) Studying cartographic heritage: analysis and visualization of geometric distortions. *Comput Graphics* 35(2):402–411
- Kogutowicz M (1902) *Teljes földrajzi atlasz. (Complete Geographic Atlas)*. Magyar Földrajzi Intézet
- Snyder JP (1985) *Computer-assisted map projection research*. U.S. Geological Survey Bulletin 1629
- Tomasi C (2013) Orthogonal matrices and the singular value decomposition. <https://www.cs.duke.edu/courses/fall13/compsci527/notes/svd.pdf>. Accessed 16 Mar 2016
- Wijewickrema S, Esson C, Papliński A (2010) Orthogonal distance least squares fitting: a novel approach. *Computer Vision, Imaging and Computer Graphics. Theory and Applications*, vol 68, pp 255–268
- Wylie CR (2011) *Introduction to projective geometry*. Dover Publications, New York