Abstract

Historical raster maps usually do not have an identified coordinate system. Due to the multitude of possibilities, the problem of selecting an appropriate reference system can be time-consuming and ineffective (especially in the case of selection of little-known systems). This study aimed to prepare a search supporting system for finding the most appropriate Coordinate Reference System (CRS) and the best possible optimization parameters for a calibrated raster map. The selection is performed by: finding control points on the raster and reference maps, transforming the reference map datum to the most promising systems, optimizing the raster map with respect to the coordinate reference system, calculating the mean-squared error (MSE) based on the shift during the transformation and sorting the results based on the error in ascending order and showing the offsets on the raster map. The results showed the effectiveness of the method. Despite the fact that this solution does not guarantee finding the appropriate mapping without additional user intervention, it significantly narrows the search scope.

Keywords: transformation parameters estimation, map calibration, coordinate reference system, software

Streszczenie

Historyczne mapy rastrowe z reguły nie posiadają zidentyfikowanego układu współrzędnych. Wielość możliwości sprawia iż problem doboru odpowiedniego układu odniesienia potrafi być czasochłonny i nieefektywny (w szczególności w przypadku doboru mało znanych układów). Celem tego badania było sporządzenie systemu wspomagającego znajdowanie najodpowiedniejszego układu współrzędnych oraz najlepszych możliwych parametrów transformacji dla kalibrowanej mapy rastrowej. Dobór jest wykonywany poprzez: znalezienie punktów kontrolnych na mapie rastrowej oraz referencyjnej, przekształcenie układu odniesienia mapy referencyjnej do najlepiej rokujących układów, optymalizacji mapy rastrowej względem kandydującego układu odniesienia, obliczenia błędu średniokwadratowego na podstawie przesunięcia podczas transformacji, posortowanie wyników w oparciu o błąd w kolejności rosnącej oraz pokazanie przesunięć na mapie rastrowej. Badanie to wykazało, iż metoda ta jest skuteczna. Rozwiązanie to pomimo tego iż nie daje gwarancji znalezienia odpowiedniego odwzorowania bez dodatkowej ingerencji użytkownika, znacznie zawęża pulę poszukiwań.

Słowa kluczowe: estymacja parametrów transformacji, kalibracja mapy, układ współrzędnych, oprogramowanie

Chapter 1

Outline of the program scope

1.1. Description of the system supporting the search of Coordinate Reference System of raster map

It is important to identify how a system of supporting search in CRS using a raster map may look like. To do so, it is required to check what are the input and output of the problem. The input could be all data provided by the application user. Surely, the required data which needs to be given by the user is a raster map. The key point is to take into account that the map provided by the user could be scanned improperly, bent or even incorrectly drawn. Potentially important inputs could also be some map metadata which could not be read or are hardly determined by the application. The output has to be a list of the most promising coordinate systems. In such a case, it could be worth to provide additional data which can help the user to decide which set is correct. Using only the computing solution without any additional user input and finding the best outcome is extremely difficult. Variety of coordinate reference systems and a high number of input parameters results in a high number of unknowns. The main goal of the program will be matching known coordinate systems to the presented historical map and supporting in choice of the most proper one.

The first problem appears in the identification of the received image as a map. Due to the fact that the picture has no coordinate system defined, it is required to find a way to acquire it. It can be done by the process of georectification. For this process, it is needed to have the same area presented in the referencing map with the identified CRS. The links between the input and reference map have to be done in at least three control points. Then some of the transformations (Helmert, polynomial, spline, projective, or adjust) have to be used. Finally, after the georeferencing process, the historical map is translated to reference map CRS parameters. The issue arises when the method of choosing the appropriate coordinate system for calculation has to be selected.

A straightforward solution seems to be iteration through the entire set of CRS, with the previous filtering out not matching coordinates.

The iterative algorithm (figure 1.1) is a method that tries to find the best possible solution with brute-force method as the reference map would be converted to every available coordinate system. In addition, the georectification would be done for any possible item. The CRS pool could be limited by: grid availability, shape of the graticule, parallel lines, meridians' convergence, if the parallels are arcs, and area of usage. Additionally, the manual indication of the type of projection by the user can reduce the search set. The results relay on the reliability of quality factor. The quality factor will be *mean square error* offset of each of the envisaged coordinates relative to the target points. The translation algorithms and global parameters will not

change during transformation, so the results should be treated as relatively reliable from that perspective.

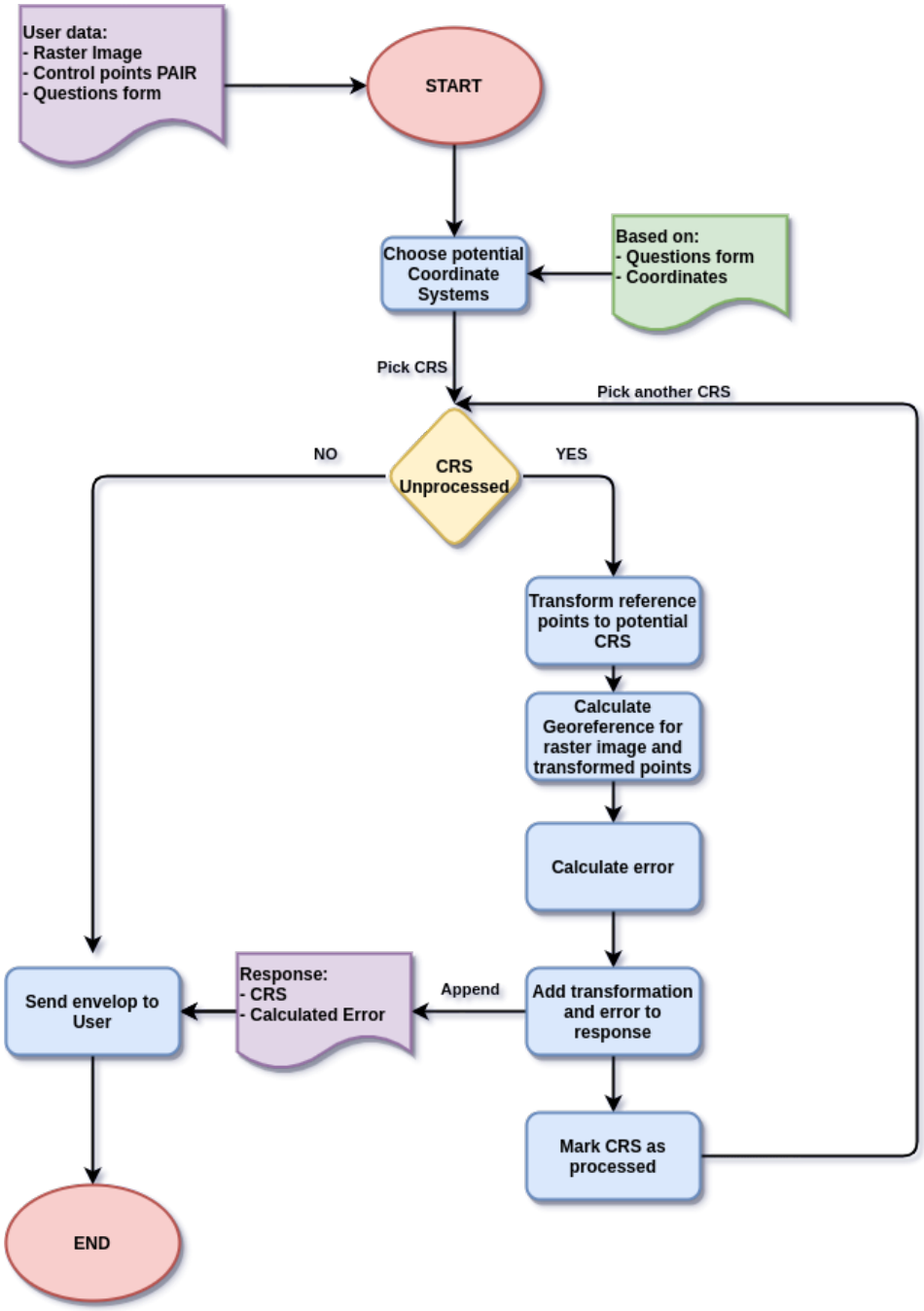

Fig. 1.1: Algorithm for supporting search of coordinate reference system

The user steps during interaction with the application have been defined (figure 1.2) for the purpose of creating the user interface. Users will have to provide details such as: historical map, input image questionnaire (to reduce the potential CRS set), and mark control points.

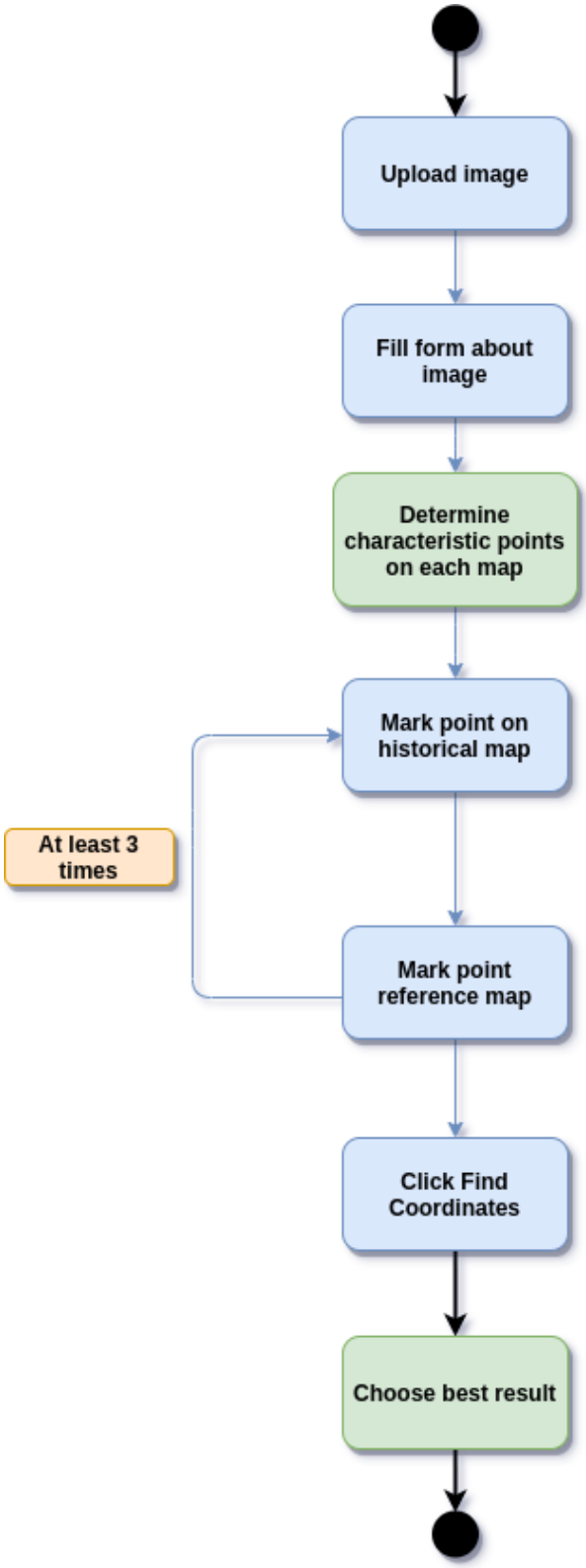

Fig. 1.2: User input procedure

1.2. Tools and technologies

To provide a platform agnostic user interface, a web application will be created. User interface part will be implemented with the use of React library as it is a popular solution nowadays and it is

well maintained. React[1] is an open source front-end library written in Javascript. The framework has been created by Facebook and initially released in 2013. The core React functionality is state management and efficient state rendering on DOM with virtual Document Object Model implementation, which updates DOM only if a visible difference appears. React offers high modularization with the support of components. Components are reusable pieces of code that contain context-specific html and JS code, which could be packed as a library and reused in any other project.

One of the main libraries used for presenting an interactive map in the project will be a **leaflet**[2], which is the leading open-source JS library for interactive maps. It offers basic map functionalities: *zooming, positioning and adding markers*. There are also a lot of community-driven map plugins such as: searching, scaling, reading points from file, adding image overlays or switching map providers as it uses *Open Street Map*[3] by default.

Konva[4] is another crucial library used for input map manipulation such as: zooming, adding markers, and positioning. It is HTML5 Canvas JS framework that enables advanced image manipulations like layering, transition, positioning, animation, and node nesting. To each drawing defined under Konva stage, it can be assigned a handler for image manipulation.

Another library, **axios**[5], will be used to communicate with API. The last of the main libraries is **Bootstrap**[6], which is used to provide a user-friendly interface.

The main part of computing will be performed on the server's side with the exposed API. Back-end side will be written in **Python**[7] language as it allows for rapid development and it is highly extensible by numerous libraries. For API exposure **Flask**[8] framework will be used. It is popular Python micro web framework initially released in 2010. It supports extensions such as: **Gunicorn**[9] application server and **flask-cors**[10] support.

Another library used for switching reference map CRS into possible CRS is **pyproj**[11]. It is wrapper interface for PROJ software which is used for transforming geospatial coordinates from one coordinate reference system to another. The last library used for parameters' optimization is **NumPy**[12]. It allows for calculations over large multidimensional arrays and matrices.

To use the application, it is necessary to run the web application server locally or externally. To publish the server externally, platforms such as **Netlify**[13] and **Heroku**[14] can be used. Netlify is a serverless hosting platform for building and deploying web applications and static websites from git repositories. It can be used to deploy front-end part of the application. Heroku is a cloud platform-based service, which allows the customer to provision, instantiate, run, and manage an application bundle without the need of infrastructure and operating system services. It could be used as a deployment platform for back-end part of the application.

[1] https://reactjs.org
[2] https://leafletjs.com
[3] https://www.openstreetmap.org/
[4] https://konvajs.org
[5] https://github.com/sheaivey/react-axios
[6] https://getbootstrap.com
[7] https://www.python.org
[8] https://flask.palletsprojects.com/en/2.0.x/
[9] https://gunicorn.org
[10] https://flask-cors.readthedocs.io/
[11] https://pyproj4.github.io/pyproj/stable/
[12] https://numpy.org
[13] https://www.netlify.com
[14] https://heroku.com

Chapter 2

Program functionality

The program has been created in the possibly most intuitive way. The user interface uses *Bootstrap* markup framework which is broadly used with web applications. Due to the fact that the application is deployed on World Wide Web, the user does not need to install the application. The only need is to enter a website with a given url address.

2.1. Application startup

Application could be used by running it locally or as an external service. Both methods require installation of the required dependencies and running the back-end and front-end. Listing 2.1 presents the procedure of running the back-end part of the application: creating *virtual environment* for Python, installing missing dependencies and running *gunicorn* application server.

Listing 2.1: Application back-end installation and run steps

```
virtualenv venv

# Source virtual environment on Linux
source venv/bin/activate

#  Source virtual environment on Windows
venv\Scripts\activate

# Install required dependencies
pip install -r requirements.txt

# Run application
gunicorn backend.app:app
```

Listing 2.2 shows procedure of starting front-end application server: entering front-end catalogue, installing dependencies, and running a server. This procedure could be done interchangeably with Node Package Manager[1] commands such as: `npm install, npm start` or Yarn Package Manager[2] with commands: `yarn install, yarn start` but preferred way is to use only one package manager in order not to mix up serving component dependencies.

Listing 2.2: Application front-end installation and run steps

```
# Enter front-end catalogue
cd front-end

# Install  dependencies
```

[1] https://www.npmjs.com
[2] https://yarnpkg.com

```
yarn install

# Start development server
yarn start
```

2.2. Providing inputs

Regardless of the way the application is launched, after entering the address (`http://localhost:3000` is default for local run), a page with two maps is displayed (figure 2.1). On the left-hand side there is an image of historical map, on the right-hand side there is a reference *Open Street Map*. Below the map on the left side there is a button with the name *"Choose File"*, at the first step the user needs to choose a raster image from the device. The chosen image should replace the left-hand side of the map.

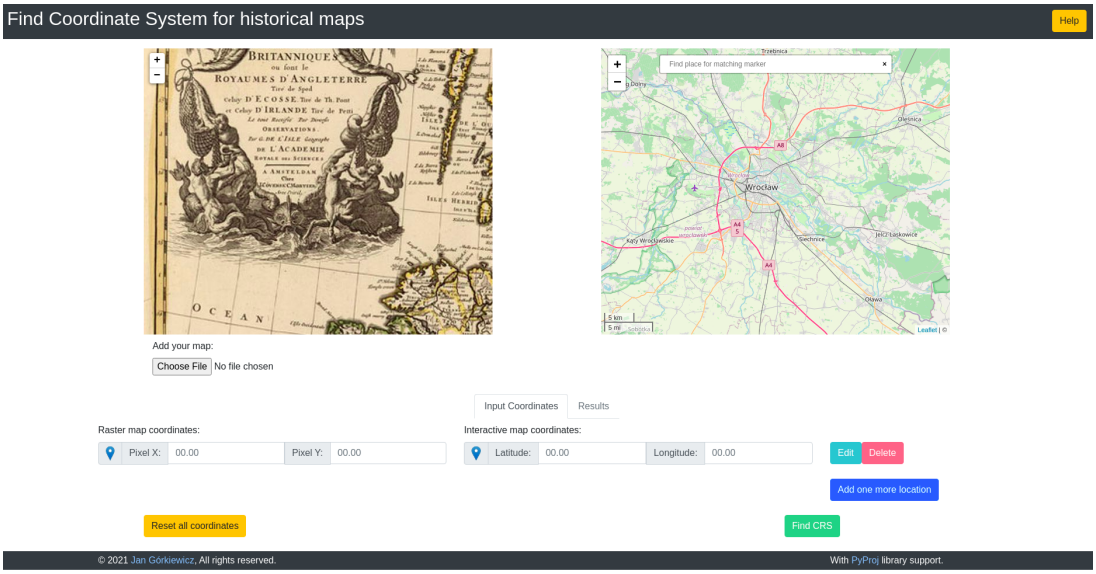

Fig. 2.1: Application main screen

Once the input map is added, the next step is to choose Ground Control Points on both: the reference and historical map. Figure 2.2 describes section called *Input Coordinates*. This section includes coordinates management form divided into two inputs. The first input group entitled *Raster map coordinates* presents pixel (Cartesian) coordinates entry for the map added by the user. Second group entitled *Interactive map coordinates* shows geographic coordinates entry for the reference map. On both sides of the form, there is a tag with which a given point will be marked on the maps. Figure 2.2 also presents five buttons:

- **Edit** – Make markers changeable on both maps with "drag and drop" method, map click method, and manual input. Only one marker could be edited at the same time. Disabled edit button and other disabled rows indicate which marker is editable.
- **Delete** – Remove the marker with coordinates.
- **Add one more location** – Add new coordinates row
- **Find CRS** – Trigger calculations for added coordinates. At least 3 pairs of points have to be provided in order to enable this button.
- **Reset all coordinates** – Remove all current results and inputs provided and return to the initial state.

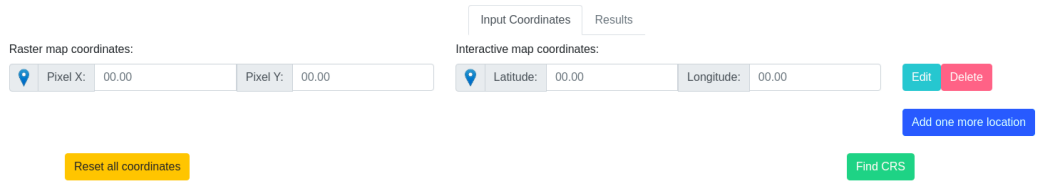

Fig. 2.2: Coordinates form initial state

There are three possible ways to add ground control points: typing manually into the form, clicking the location on the map, or "drag and drop" already provided state. To add a tag, it is required to locate a pair of control points on both maps. To add a point, the user has to click on the map (figure 2.3). Zoom can be used to achieve better precision. Additionally, to find the location in the interactive map, the location search toolbar can be used.

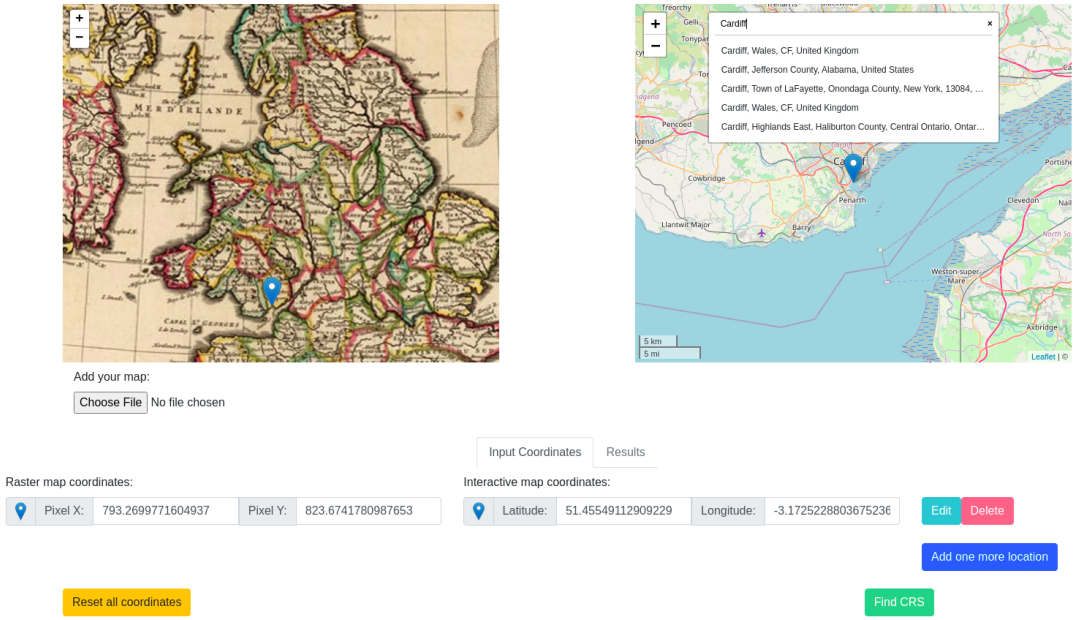

Fig. 2.3: Added matching GCP

Figure 2.4 presents the created ground control points. It is required to add at least three GCP's to calculate the transformation parameters. Higher number of coordinates pairs helps to reduce the impact of inaccurately positioned control points. In case of accidental website closing and reentering application, the input data should be automatically retrieved as the markers are persisted with cookies. In case of all ground control points are properly filled, the user needs to click a button with title "**Find CRS**". This results in receiving a notification that CRS processing has been started and in being asked for patience. After performing the calculations, the results are being displayed.

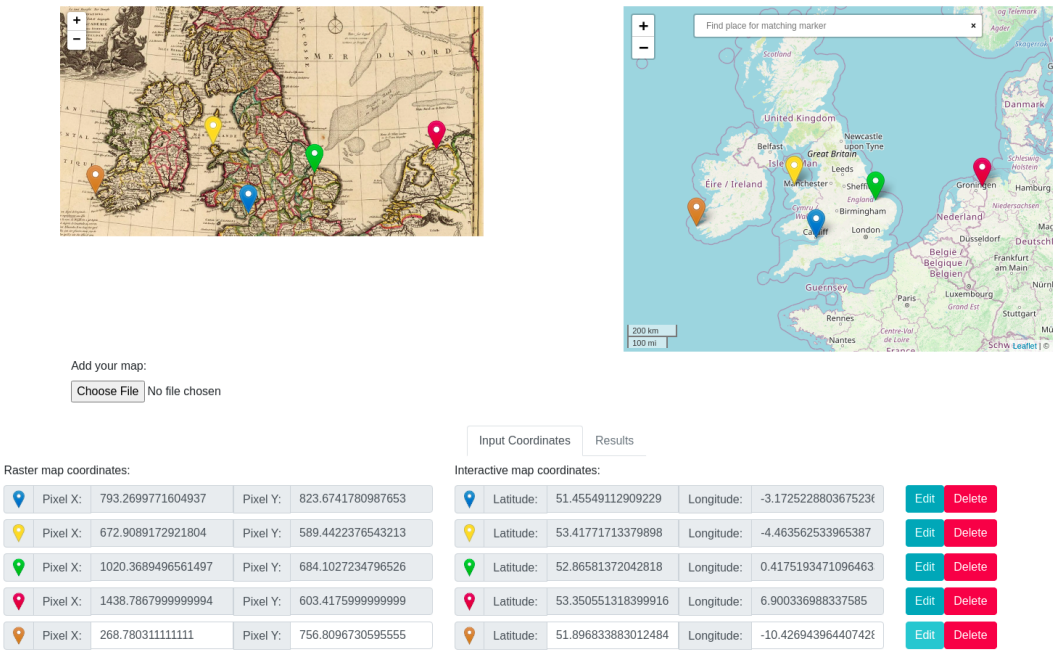

Fig. 2.4: Application with filled coordinates for ready to start calculation

2.3. Transformation calculations

The computational part is performed on the back-end using an application written in python.

2.3.1. Provided inputs

The listing 2.3 consist of a sample payload sent to the back-end part of the application. The default structure consists of either a markers' array or an interactive map bounds list. The `markers` array contains the map coordinates in respective order: `x, y` for pixels and `latitude,longitude` for the reference map. The `interactiveMapBounds` array contains the search area for coordinate systems, based on the most constrained coordinates. The parameters occur in order that the bounding box cover: south latitude, west longitude, north latitude, and east longitude, respectively.

Listing 2.3: Example of request payload sent to the API for calculations

```
{
  'markers': [
    {
      'inputMap': [
        793.2699771604937,
        823.6741780987653
      ],
      'interactiveMap': [
        51.45549112909229,
        -3.1725228803675236
      ]
    },
    {
      'inputMap': [
        672.9089172921804,
        589.4422376543213
      ],
```

```
      'interactiveMap': [
        53.41771713379898,
        -4.463562533965387
      ]
    },
    {
      'inputMap': [
        1020.3689496561497,
        684.1027234796526
      ],
      'interactiveMap': [
        52.86581372042818,
        0.41751934710964633
      ]
    }
  ],
  'interactiveMapBounds': [
    51.45549112909229,
    -4.463562533965387,
    53.41771713379898,
    0.41751934710964633
  ]
}
```

2.3.2. Selection of CRS candidates

Once Python component receives a request, it checks the validity of the payload. In case of a valid payload, a list of applicable CRS' for a given map area is loaded. The calculated list is based on the Json file covering data about all CRS' available. Json file is the dump of the query shown in the listing 2.4 over GeoRepository database[3]. Table 2.1 shows random rows sample data, received by querying the database. The *EPSG* and *boundaries* are the required data needed for filtering out the area. The rest of the data has an informative purpose.

Listing 2.4: Sql query for gathering CRS list with bounding boxes and additional details

```
SELECT DISTINCT ON (coord_ref_sys_code)
                'EPSG:' || coord_ref_sys_code::text AS EPSG,
                coord_ref_sys_name                  AS name,
                coord_ref_sys_kind,
                bbox_south_bound_lat,
                bbox_west_bound_lon,
                bbox_north_bound_lat,
                bbox_east_bound_lon,
                CRS.remarks                         AS details,
                extent_name                         AS area
FROM epsg_coordinatereferencesystem AS CRS
         INNER JOIN epsg_usage USAGE ON CRS.coord_ref_sys_code = USAGE.↘
             ↪ object_code
         INNER JOIN epsg_extent EXTENT on USAGE.extent_code = EXTENT.↘
             ↪ extent_code
WHERE CRS.deprecated = 0
  AND USAGE.object_table_name = 'epsg_coordinatereferencesystem'
ORDER BY coord_ref_sys_code;
```

[3] sourced in https://epsg.org

Tab. 2.1: Random 20 results of query from listing 2.4. (Due to large size of the columns *coord_ref_sys_kind*, *details* have been skipped)

epsg	name	south lat	west lon	north lat	east lon	area
EPSG:2922	NAD83(HARN) / Utah Central (ft)	38.49	-114.05	41.08	-109.04	USA – Utah – SPCS – C
EPSG:2923	NAD83(HARN) / Utah South (ft)	36.99	-114.05	38.58	-109.04	USA – Utah – SPCS – S
EPSG:2924	NAD83(HARN) / Virginia North (ftUS)	37.77	-80.06	39.46	-76.51	USA – Virginia – SPCS – N
EPSG:2925	NAD83(HARN) / Virginia South (ftUS)	36.54	-83.68	38.28	-75.31	USA – Virginia – SPCS – S
EPSG:2926	NAD83(HARN) / Washington North (ftUS)	47.08	-124.79	49.05	-117.02	USA – Washington – SPCS83 – N
EPSG:2927	NAD83(HARN) / Washington South (ftUS)	45.54	-124.4	47.61	-116.91	USA – Washington – SPCS83 – S
EPSG:2928	NAD83(HARN) / Wisconsin North (ftUS)	45.37	-92.89	47.31	-88.05	USA – Wisconsin – SPCS – N
EPSG:2929	NAD83(HARN) / Wisconsin Central (ftUS)	43.98	-92.89	45.8	-86.25	USA – Wisconsin – SPCS – C
EPSG:2930	NAD83(HARN) / Wisconsin South (ftUS)	42.48	-91.43	44.33	-86.95	USA – Wisconsin – SPCS – S
EPSG:2931	Beduaram / TM 13 NE	12.8	7.81	16.7	14.9	Niger – southeast
EPSG:2932	QND95 / Qatar National Grid	24.55	50.69	26.2	51.68	Qatar – onshore
EPSG:2933	Segara / UTM zone 50S	-1.24	116.72	0	117.99	Indonesia – Kalimantan – Mahakam delta
EPSG:2935	Pulkovo 1942 / CS63 zone A1	41.37	39.99	43.59	43.04	Asia – FSU – CS63 zone A1
EPSG:2936	Pulkovo 1942 / CS63 zone A2	38.87	43.03	43.05	46.04	Asia – FSU – CS63 zone A2
EPSG:2937	Pulkovo 1942 / CS63 zone A3	38.38	46.03	42.1	49.04	Asia – FSU – CS63 zone A3
EPSG:2938	Pulkovo 1942 / CS63 zone A4	37.89	49.03	42.59	51.73	Asia – FSU – CS63 zone A4
EPSG:2939	Pulkovo 1942 / CS63 zone K2	41.15	49.26	51.77	52.27	Asia – FSU – CS63 zone K2
EPSG:2940	Pulkovo 1942 / CS63 zone K3	41.46	52.26	51.79	55.27	Asia – FSU – CS63 zone K3
EPSG:2941	Pulkovo 1942 / CS63 zone K4	41.26	55.26	51.14	58.27	Asia – FSU – CS63 zone K4
EPSG:2942	Porto Santo / UTM zone 28N	32.35	-17.31	33.15	-16.23	Portugal – Madeira archipelago onshore
EPSG:2943	Selvagem Grande / UTM zone 28N	29.98	-16.11	30.21	-15.79	Portugal – Selvagens onshore

2.3.3. Converting points between Coordinate Systems

List of coordinates from the reference map and CRS candidates list is passed to the method, which is responsible for converting points from the reference map CRS (WGS 84 / EPSG:4326) to the coordinate candidate. For the conversion purpose *pyproj* library is used. Since there is no common method to transform between coordinate systems, pyproj have predefined operations needed to transform between two different Coordinate Reference Systems. For conversion between two CRS system, *transformer* component of proj is used. The conversion task is the most CPU expensive one. Filtration CRS by area allows to decrease number of candidate coordinate system from thousands to hundreds. To solve this drawback, either downsizing of the CRS list or task parallelization could be used. Due to the fact that reducing CRS list may be very complicated and can decrease user experience, parallelization was taken into account. Table 2.2 shows the improvement in time processing for different CRS list sizes. The results show that even for list of 5 items it is worth to parallize the *proj* conversion. During the test, 8 threads have been used.

Tab. 2.2: Standard and parallelisation method processing time comparison (4 coordinates input array)

Trial	Synchronised (77 CRS-s)	Parallelised (77 CRS-s)	Synchronised (5 CRS-s)	Parallelised (5 CRS-s)
1	36.622	8.229	3.489	2.073
2	35.665	8.198	3.475	1.949
3	36.926	8.539	3.436	1.959
AVG	36.404	8.322	3.467	1.994

2.3.4. Estimation methods

The next stage of the calculation is the estimation of raster map transformation parameters such as: scale, rotation, or distortions during the digitization process. Based on the pixel points provided in the raster and in the converted reference map, an estimation of the transformation parameters needed for georectification can be provided. One of the transformation parameters estimation methods implemented is two dimensional *Helmert* transformation. In that case, only the *x, y* coordinates are changed. The algorithm implementation 2.5 is an interchanged version of equation (**??**). At first the gravity center and the difference between points for both input

and referenced maps have been determined. Another step is to determine $\sin$ and $\cos$ related coefficients presented in equation (2.1). Where C is cosine related and S sine related coefficient.

$$C = \frac{W_1}{W}, \quad S = \frac{W_2}{W} \tag{2.1}$$

where:
$W = \sum_{i=1..n}(x_i^2 + y_i^2)$
$W_1 = \sum_{i=1..n}(X_i \cdot x_i + Y_i \cdot y_i)$
$W_2 = \sum_{i=1..n}(X_i \cdot y_i - Y_i \cdot x_i)$
x_i, y_i – differences from the center of gravity for the original coordinate system (reference map)
X_i, Y_i – differences from the center of gravity for the actual coordinate system (pixel map)
C, S – cosine, sine related coefficients

Based on the calculated coefficients, it is possible to calculate the rotation matrix, scale, and shifts relative to the center of gravity (2.2). It is also possible to estimate the corrected coordinates (2.3).

$$\begin{aligned} \text{scale} &= \sqrt{S^2 + C^2}, \\ \text{Rotation} &= \begin{bmatrix} C/\text{scale} & S/\text{scale} \\ -S/\text{scale} & C/\text{scale} \end{bmatrix}, \\ \text{shift}_x &= X_0 - C * x_0 - S * y_0, \\ \text{shift}_y &= Y_0 + S * x_0 - C * y_0, \end{aligned} \tag{2.2}$$

$$\begin{aligned} X' &= X_0 + C \cdot x_i + S \cdot y_i, \\ Y' &= Y_0 - S \cdot x_i + C \cdot y_i, \end{aligned} \tag{2.3}$$

where:
X_0, Y_0 – gravity center for the actual coordinate system (pixel map)
x_i, y_i – differences from the center of gravity for the original coordinate system (reference map)
C, S – cosine, sine related coefficients
X', Y' – predicted coordinates

Based on the new predicted points in the reference map, the shift vector can be calculated. The results could be used for manual CRS identification and quality factor calculation as well [1].

Listing 2.5: Implementation of four parameters Helemert coefficients estimation algorithm.

```
1  def estimation_helmert_four(gcps_array):
2      gcps_number = len(gcps_array)
3      X0, Y0, x0, y0 = np.average(
4          gcps_array, 0)
5
6      x = gcps_array[:, 2]
7      y = gcps_array[:, 3]
8      X = gcps_array[:, 0]
9      Y = gcps_array[:, 1]
10
```

```
    xi = x - x0
    yi = y - y0
    Xi = X - X0
    Yi = Y - Y0

    s_numerator, c_numerator, denominator = 0, 0, 0
    for i in range(gcps_number):
        c_numerator += Xi[i] * xi[i] + Yi[i] * yi[i]
        s_numerator += Xi[i] * yi[i] - Yi[i] * xi[i]
        denominator += xi[i] * xi[i] + yi[i] * yi[i]

    c_factor = c_numerator / denominator  # scale, rotation factor
    s_factor = s_numerator / denominator  # scale, rotation factor

    rotation = rad2deg(arctan(s_factor / c_factor))
    scale = sqrt(s_factor * s_factor + c_factor * c_factor)
    shift_x = X0 - c_factor * x0 - s_factor * y0
    shift_y = Y0 + s_factor * x0 - c_factor * y0

    pred_x = X0 + c_factor * xi + s_factor * yi
    pred_y = Y0 + c_factor * yi - s_factor * xi

    shift_vector_x = pred_x - X
    shift_vector_y = pred_y - Y

    return pred_x, pred_y, shift_vector_x, shift_vector_y, {↘
        ↪ TRANSFORMATION_HELMERT_FOUR_CONST: {
        "rotation (deg)": rotation,
        "scale": scale,
        "shift_x": shift_x,
        "shift_y": shift_y
    }}
```

Another algorithm is the estimation of polynomial transformation parameters for the raster map. Listing 2.6 presents the algorithm of first, second, and third order polynomial coefficients' estimation. The first-order polynomial (2.4) is for affine transformations. When the dataset must be bent or curved, the usage of second (2.5) or third (2.6), or higher order polynomial is needed. Higher order polynomials tend to give extrapolation errors, lower order ones tend to give random errors. Polynomial regression model is developed by using the least squares approximation. The least squares method aims to minimise the mean distance between the estimated values and expected from the dataset. The minimum number of points to be provided should be $n = o + 1$ where o is a polynomial order. However, a higher number of points could improve the polynomial fitting. [2]

$$\begin{aligned} X &= a_0 + a_1 x + a_2 y \\ Y &= b_0 + b_1 x + b_2 y \end{aligned} \tag{2.4}$$

$$\begin{aligned} X &= a_0 + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2 \\ Y &= b_0 + b_1 x + b_2 y + b_3 xy + b_4 x^2 + b_5 y^2 \end{aligned} \tag{2.5}$$

$$\begin{aligned} X &= a_0 + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2 + a_6 x^3 + a_7 x^2 y + a_8 xy^2 + a_9 y^3 \\ Y &= b_0 + b_1 x + b_2 y + b_3 xy + b_4 x^2 + b_5 y^2 + b_6 x^3 + b_7 x^2 y + b_8 xy^2 + b_9 y^3 \end{aligned} \tag{2.6}$$

where:

X – raster map pixel Cartesian x coordinate

Y – raster map pixel Cartesian y coordinate
x – Cartesian x coordinate of candidate CRS (from reference map)
y – Cartesian y coordinate of candidate CRS (from reference map)
$a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9$ – X Transformation unknown parameters
$b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9$ – Y transformation unknown parameters

Listing 2.6 presents the implementation of the parameters' estimation algorithm for polynomial transformation. First step of the estimation function is the preparation of polynomial order (from 1 to 3). Then the polynomial regression model is solved over the reference coordinates with least-squares method. Finally, the dot product of the solved matrix results with the predicted coordinates. Shift is calculated as a subtraction between the estimated and pixel map coordinates. The shift is used for mean square error calculations.

Listing 2.6: Estimation of polynomial transformation coefficients.

```
1  def estimation_polynomial(order, gcps_array):
2      gcps_number = len(gcps_array)
3      img_x = gcps_array[:, 0]
4      img_y = gcps_array[:, 1]
5      ref_x = gcps_array[:, 2]
6      ref_y = gcps_array[:, 3]
7      if order == 1:
8          # X = a0 + a1x + a2y
9          # Y = b0 + b1x + b2y
10         matrix = np.zeros((gcps_number, 3), dtype=np.float)
11         matrix[:, 0] = 1
12         matrix[:, 1] = ref_x
13         matrix[:, 2] = ref_y
14     elif order == 2:
15         # X = a0 + a1x + a2y + a3xy + a4x^2 + a5y^2
16         # Y = b0 + b1x + b2y + b3xy + b4x^2 + b5y^2
17         matrix = np.zeros((gcps_number, 6), dtype=np.float)
18         matrix[:, 0] = 1
19         matrix[:, 1] = ref_x
20         matrix[:, 2] = ref_y
21         matrix[:, 3] = ref_x * ref_y
22         matrix[:, 4] = ref_x * ref_x
23         matrix[:, 5] = ref_y * ref_y
24     elif order == 3:
25         # X = a0 + a1x + a2y + a3xy + a4x^2 + a5y^2 + a6x^3 + a7x^2y + ↘
               ↪ a8xy^2 + a9y^3
26         # Y = b0 + b1x + b2y + b3xy + b4x^2 + b5y^2 + b6x^3 + b7x^2y + ↘
               ↪ b8xy^2 + b9y^3
27         matrix = np.zeros((gcps_number, 10), dtype=np.float)
28         matrix[:, 0] = 1
29         matrix[:, 1] = ref_x
30         matrix[:, 2] = ref_y
31         matrix[:, 3] = ref_x * ref_y
32         matrix[:, 4] = ref_x * ref_x
33         matrix[:, 5] = ref_y * ref_y
34         matrix[:, 6] = ref_x * ref_x * ref_x
35         matrix[:, 7] = ref_x * ref_x * ref_y
36         matrix[:, 8] = ref_x * ref_y * ref_y
37         matrix[:, 9] = ref_y * ref_y * ref_y
38
39     least_squared_reference_x = np.linalg.lstsq(matrix, img_x, rcond=↘
           ↪ None)
40     least_squared_reference_y = np.linalg.lstsq(matrix, img_y, rcond=↘
           ↪ None)
```

```
41
42      pred_x = matrix.dot(least_squared_reference_x[0])
43      pred_y = matrix.dot(least_squared_reference_y[0])
44
45      shift_vector_x = np.array(pred_x) - img_x
46      shift_vector_y = np.array(pred_y) - img_y
47
48      return pred_x, pred_y, shift_vector_x, shift_vector_y, {
49          TRANSFORMATION_POLYNOMIAL_CONST:
50              {
51                "order": order,
52                "least_squared_x": least_squared_reference_x[0].tolist(),
53                "least_squared_y": least_squared_reference_y[0].tolist()
54              }
55      }
```

In the last part, the quality factor is calculated. It is needed to determine which coordinate system, with what parameters is the best match for the raster map. Mean square error is the sum of vector shifts between the pixel map and the estimated coordinates. MSE is calculated per each coordinate from CRS candidate list. The lower MSE possibly means better CRS fitting, which suggests that the obtained coordinate system may be the one in which the raster map is created. To avoid negative values, shifts are squared. Listing 2.7 presents the implementation of means square error formula.

Listing 2.7: Implementation of calculating mean square error of shift vector

```
1  def mean_square_error(shift_vector_x, shift_vector_y, gcps_number):
2      squared_vector_sum = 0
3
4      shift_vector = np.sqrt(shift_vector_x * shift_vector_x +
           ↪ shift_vector_y * shift_vector_y)
5
6      for i in range(gcps_number):
7          squared_vector_sum += shift_vector[i] * shift_vector[i]
8
9      return squared_vector_sum / gcps_number
```

Finally, all calculated CRS candidates are returned to the front-end part of the application.

2.4. Results analysis

Returned results received by the web application are presented under the results panel. The main part of the results panel, shown in figure 2.5, is a table with results. The table consists of the following columns:

- **Name** – Common coordinate system name
- **EPSG** code
- **MSE** – Mean square error of shift vector. CRS matching quality factor.
- **Converted Points** – Converted points from the reference map to candidate CRS
- **PredX**, **PredY** – Predicted position array for candidate CRS
- **ShiftX**, **ShiftY** – Shift array calculated from the results of transformation method
- **Parameters** – Estimated parameters, values depend on transformation type
- **Area** – Area of specific CRS use
- **Details** about Coordinate System

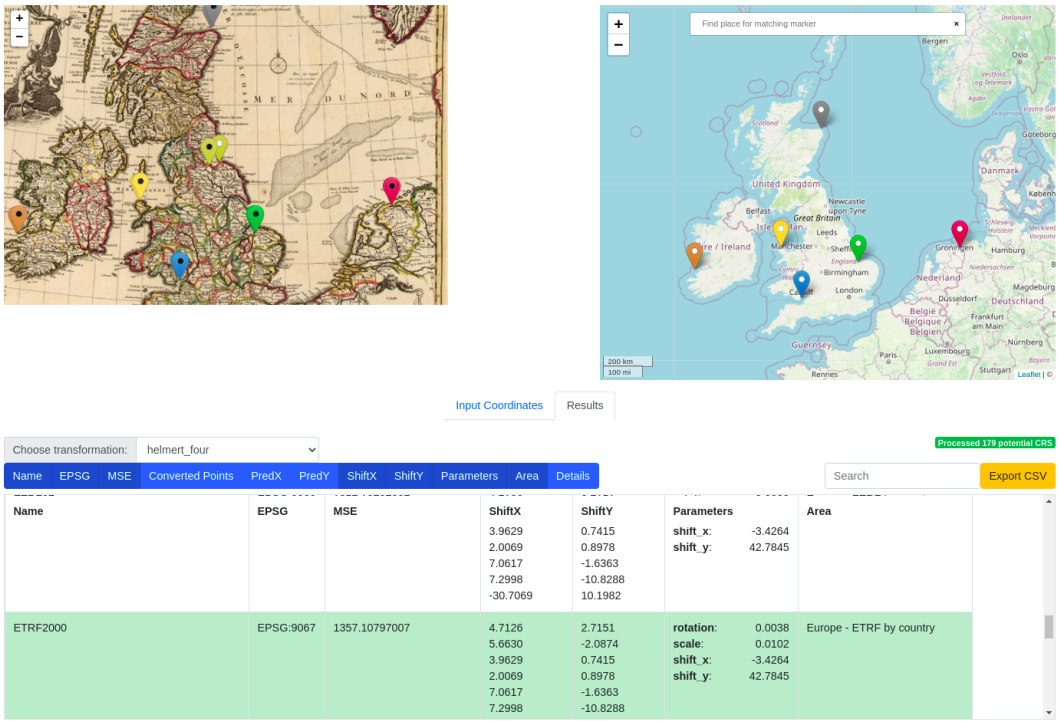

Fig. 2.5: Calculated Coordinate Systems and its parameters

Calculated result table is sorted in ascending order over the mean square error column. Based on that information and other provided details, the user should be able to determine in which CRS raster map is created. There is a possibility to validate the coordinate shift visually. To do so, the user can click the specific row on the list. The selected CRS coordinates appear in the raster map with the same colors of markers. Only the middle marker is black instead of white for the transformed points. Based on the visual preview and information in the table, the user should be able to determine the proper CRS. Above the table, there is a navigation panel, which allows to choose a calculation for a specific estimation method for transformation such as four parameters' Helmert or polynomial transformations. Another navigation option is the toggle of a specific column, to make the table clearer. User is also able to search through all columns in case of the need to reduce the scope of CRS candidates.

References

[1] R. Kadaj. Zasady zastosowania metody transformacyjnej do przeliczeń punktów z układu „1965” lub lokalnego do układu „2000”, 2006.

[2] neutrium.net. The Neutrium website. fitting of a polynomial using least squares method. https://neutrium.net/mathematics/least-squares-fitting-of-a-polynomial/, 2015.