TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Environmental Sciences   Institute for Cartography

Master Thesis

A Contribution to Computer-Assisted Reconstruction of Selected Line Features from Scanned Maps

submitted by **Jiaqin Ni**

born on 20.04.1993 in Shanghai

submitted for the academic degree of
Master of Science (M.Sc.)

Submission on	04/10/2017
Supervisors	Dr. Nikolas Prechtel TU Dresden – Institute for Cartography
	Prof. Dr. Markus Wacker HTW Dresden
	Dr. Corné van Elzakker
	University of Twente – ITC

Original Task Specification

Objective

A theory part shall summarise technical approaches of automated line reconstruction from maps and similar documents. It is in general completely legitimate to confine the concept and the implementation of feature extraction in this thesis to one specific group of linear map elements (as rivers, roads, etc.).

It is common knowledge that due to graphic conflicts (overlays and touching of elements) within a map face vector paths (streets, rivers, railways, etc.) are frequently intermittent or not easily separable. Useful results (connected, well-tailored geo-features) cannot be achieved by a simple vectorisation. A further task relates to a reduction of dual line signatures or graphically augmented lines (e.g. railway signatures) to their centre line, or, the contrary, a connection of dashed line signatures to contiguous lines. In order to find good reconstruction techniques, a concise scheme of cartographic lines in (historic) maps shall be sought. A comprehensive feature space embedding e.g. parameters of line shape, line width, line fillings and line colour may thus be offered in order to pinpoint processing of the scan onto a specific graphic expression. Especially in respect to colour space, variations due to imperfect print techniques, the aging of the document (e.g. stains), and the mixing of foreground/background signals (pixels) along feature edges have to be considered. The concept for an implementation may concentrate on modules taken from Open Source libraries. Python will be the preferred script language, both for the existence of specialised packages and compatibility with geo-software. Examples: OpenCV for Python offers quite a range of image manipulation and computer vision functionality. For a vectorisation of original or pre-segmented images, modules like "Potrace" (http://potrace.sourceforge.net/#description) or "mapseg" (http://wiki.openstreetmap. org/wiki/Mapseg) may be tested. The final choice of suitable software modules will, however, be completely a decision of the author.

The thesis shall contribute to a proposed generic toolbox dedicated to selective computer-assisted map feature extraction. "Toolbox" indicates a preference for specialised, efficient, upgradable compact programme modules over a monolithic programme solution tackling the complete extraction task. Human interaction is welcome and promising quicker results, which are crucial within a web service. Activities can be confined to the map face. Statistics specifying the extraction quality and the performance shall accompany the implementation. A summarising chapter shall critically evaluate the achievements and name priorities in further work in the context of map feature extraction.

Statement of Authorship

Herewith I declare that I am the sole author of the thesis named

„A Contribution to Computer-Assisted Reconstruction of Selected Line Features from Scanned Maps“

which has been submitted to the study commission of geosciences today.

I have fully referenced the ideas and work of others, whether published or unpublished. Literal or analogous citations are clearly marked as such.

Dresden, 04/10/2017 Signature [illegible]

Contents

Figures

1 Introduction

In the age of digitisation and internet communication, an increasing number of historical maps has been scanned and archived as high-resolution digital images. This opens the door to view these documents, and to visually compare them to others (in order to compare the geographic knowledge of the time or the spatial focus of the authors), or to reveal changing spatial knowledge in a culture, or to detect historic spatial developments over time. However, all these scanned historic sources will not enable a scholar to arrive at unambiguous quantitative measures in their descriptions and comparisons as long as the image of a map stays as an image and will not be transformed into structured and individually processable geo-features in the sense of historic GIS data. This transformation, however, is a giant task and will need a high degree of automation to show a substantial progress. This master thesis tries to contribute to this field of research in concentrating on linear features within historic maps and in trying to find ways to arrive at structured vector features without the simple solution of manual digitising.

A possible solution should be provided to implement computer-assisted line feature detection, extraction and reconstruction during the research. The work aims at a major contribution to a generic toolbox, which performs a complete process of detection, separating, extraction and vectorisation. Various functions should be embedded as modules of the toolbox, which are able to perform individual steps in line feature extraction. The research should focus on a specific group of line feature (e.g. roads, streets, rivers, etc.), instead of an attempt of extracting line features from the scanned map in an unspecific way.

This research could be of great interest to anyone who deals with feature vectorization of scanned maps, especially historical maps.

2 Related Work

Research in the field of feature recognition and extraction from paper maps has been going on for many years, in addition that nowadays the well-developed scanning technology allows paper maps to be scanned in higher resolution and quality, therefore, large amount of previous related works and publications could be found in this field of study. Generally speaking, a typical digital feature extraction procedure should include the following phases: (1) digitization of the original paper-based material; (2) filtering, or foreground and background separation; (3) thresholding; (4) thinning and trimming the features; (5) vectorization. (Salvatore, Guitton, 2004)

In this section, the related works will be spread into several related sections and introduced individually.

2.1 Layer Separation

Layer separation is in respond to the second phase: filtering. As the quality of scanned maps usually vary greatly, especially historical maps, since the quality of the map itself could be affected by lots of external factors. For instance, imperfect archaic printing technique, different colouring method and style, or even stains on the map documents could largely affect the process of determining continuous line features. In order to have a clear view of the line features that need to be extracted, the negative influence of the noise and disturbance should be minimized. In the preliminary stages of processing, the foreground and background of the scanned image should be separated. The foreground indicates the features that are needed to be extracted, while the background is in represent of other needless features in the image. Zack et al. created a global search threshold (THR) in the intensity histogram, using the normalized height and dynamic range of the histogram, which effectively contributed in foreground pixel segmentation (1977). Besides, reducing the number of colours could also be used to help layer segmentation. Chiang and Knoblock used Mean-shift and K-means algorithms to merge colours and to limit the eventual number of the colours in the output image (2009). For some images that obtain wide variations in colour intensity, the K-means algorithm could be combined with a previous image enhancement to achieve better results (Dhar and Chanda, 2006). Cao and Tan proposed an algorithm to recognize certain pixel as a black pixel and utilized a morphological method to remove the black layer in the pre-processing stage (2001). A binarization step could also be taken to first extract the foreground pixels. In some cases, the binarization step could be combined with a morphological closing in order to first classify the colour pixels and then remove small line features (Pouderoux et al., 2007).

2.2 Line Feature Identification

The third and fourth phases could be concluded into the process of line feature identification. In different cases of various kind of maps, corresponding methods should be taken to identify different kinds of line features from assorted backgrounds. Most common cases are normal topographic maps (however not necessarily historical maps). Usually a colour segmentation method is used to identify cartographic features according to certain mapping colour standards. Pouderoux et al. separate the image into five layers according the RGB value in each pixel. In the blue layer, which indicates rivers and other hydrological features, a recognition procedure based on predefined decision criteria were combined with the boundary extraction in order to extract the blue features (2007). In handling publishes with standard map keys, extracting certain colours could effectively remove one kind of features. Khotanzad et al. constructed a colour key to overcome the problem of false colours and colour aliasing in extraction of contour lines from USGS topographical maps (Khotanzad, Zink, 2003).

When dealing with colour extraction, the selection of colour space requires careful consideration. In the research of Salvatore and Guitton, the backwards of using RGB colour space was pointed out (2004). A colour space with improved perceptual uniformity and tractability was suggested, in their case, the HSV colour space. This separation between hue and other colour properties makes it easier for users to control the colour shade only by values. Besides topographic maps, other basic features that are included in various maps could also be regarded as clues to feature extraction or subtraction. For instance, detecting the possible Hough lines could emphasize benefits to recognizing road pixels (Chiang, Knoblock, 2009). This is less affected by the colour of the map but more concentrate on the line feature itself. Other methods include double-line format checking and parallel-pattern tracing (Chiang et al., 2005). These traditional but effective methods focus on the profile of line features. Considering either single or double line shape is popularly used to represent roads and streets even in historical maps, these methods could be potential candidates for road recognition in this project work.

2.3 Feature Thinning Methods

A pre-extracted foreground need to be trimmed before vectorization. Liu and Dov concluded several thinning based methods in the image based approach (1999), which is a process of applying morphological operations to the image. It is intended to return with a set of black pixels as the skeleton (Montanari, 1969), forming a clear topological structure of the input image, making the image much easier to analyse and operate in later stages.

During the morphological stage, making use of a series of operations could help to remove the outer layer of the contour lines in order to enhance the main structure of the image, and remove small noise pixels as well. In the work of Chiang et al., a generalized dilation operator, a generalized erosion operator, and a thinning operator were applied in order (2005). The hit-or-miss transformations was performed using 3-by-3 binary masks to scan the input image and measure whether the mask match the input image or not. This operation would return a result with either “hit” or “miss”, resulting in first an enhancement and then a thinning effect of the line features. When extracting the contour lines from scanned topographic maps, Salvatore and Guitton also applied a thinning procedure to reduce the width of the output line features (2004). In order to achieve better results and get rid of non-interested features, a smoothing process was applied after the thinning procedure.

There are different algorithms proposed for thinning operation, and could be used in various approaches such as OCR or line recognition (Liu, Dov, 1999). These include both complex algorithms and straightforward methods, even some built-in handy operations. In this project, some of the appropriate methods were tested and applied to the map image for line extraction.

3 Data and Environment

The historical maps used in the extraction process were all taken from the Virtuelles Kartenforum 2.0 (Virtual Map Forum 2.0), a crowdsourcing approach built for georeferencing of the historical maps taken from the main part of the SLUB collection. The map collection at SLUB focuses on maps of Saxony and topographical maps of Germany and Europe during the 17th to the 20th century, including both historical maps and their reproductions. It is one of the oldest and largest map collection in Germany. The total number of collection has already grown up to 177,000 within the past 10 years ("SLUB Dresden: Maps", 2016). As one of its portals, the Virtuelles Kartenforum 2.0 provides a platform for average users to contribute to the large amount of manual georeferencing for the old maps from the collection. So far, over 5,600 georeferenced historical maps could be found in high resolution, all of which are saved in tiff uncompressed format ("SLUB Dresden: The Map Collection", 2016).

Maps in the SLUB collection are composed of various types such as geological maps, hydrographic maps, country maps, fortress plans, maps of residential areas, as well as Milestones of Saxony (Meilenblätter von Sachsen), the result of the topographical land survey of Saxony carried out between 1780 and 1806. The maps selected for this project work were published during the 18th to the middle 19th century from the collection. The scale of the test maps are mainly between 1: 12,000 to 1: 8,000, which indicate that the test maps are all large-scale maps for towns and its surrounding area. The test maps include both topographic maps and detail plans. The cartographic techniques for making these historical maps could be divided into two categories, one of which is lithography and the other is hand painting and colouring.

These test maps cover a variety of terrain representations and as well, a relatively wide range of feature complexity. Being regarded as testing maps, these maps in the collection should share some basic features in common while having individual characteristics. One of the most recognizable features of these historical maps should be the hachures. As an old technique of representing relief, which was standardized in 1799, hachures account for a significant percentage of shading technique in these historical maps. On the other hand, the test maps are all manually water coloured, and as a result, the features are not evenly painted with a united saturation level, as well as the mixing of foreground and background features. In addition to the uneven colouring quality, the colour in all these maps started to become yellowish as time goes by, accompanied with irregular stains. It is also problematic that the ancient fonts used in these historical maps are different from modern serif or sans-serif letters.

The original downloaded map data are in tiff format, with a resolution of 72 dpi. In order to have batter control of the data volume, the test maps were resized before any processes. The sizes of the maps were adjusted into 4000*4000 pixels, without changing the original resolution, and saved in jpg format.

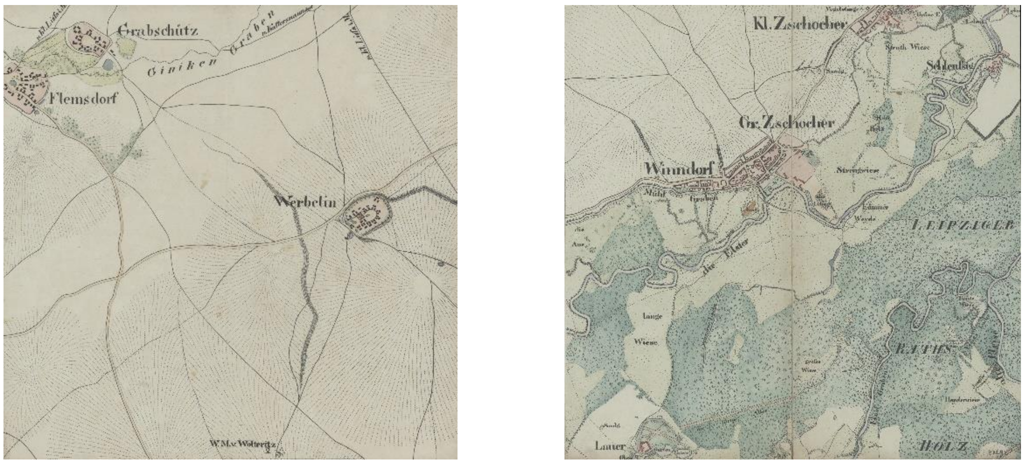


Figure 3.1 Preview of test maps.

Figure 3.1 a shows an example of the test maps that include basic line structures and limited cartographic features. The foreground is mainly composed of roads, cartographic icons, small clusters indicating buildings in residential area, and texts, which are mainly names of towns and rivers. The background includes the colour of the base map, the areal features, and hachure lines, which are indeed representative as cartographic features in historical maps during 18th to 19th century. The roads were sorted into three categories in this test map, namely main road, street, and walking paths, each with its corresponding road profile.

While Figure 3.1 b represents a test map with rather abundant number of features, with large percentage of background areal features and almost all kinds of foreground features. As it can be observed from the figure below, most of the map is covered by vegetation (grasslands or forests), in terms of dark and light green colouring and densely distributed small black icons. In addition, various foreground features are also included in this map example, especially with curved and sinuous line features, hence it could be regarded as a rather integrated map example.

This project work requires a Python working environment. The Python version used is 2.7.13. Besides, many extra packages are applied in this work, especially those benefit the image processing procedure. In addition, GIS software, third-party packages, and graphic user interfaces that use other programming language are also applied only if they are suitable for the case.

4 Workflow

4.1 Image Properties

Before actually diving into the extraction work, it is necessary to take a deep look into the properties of the test maps that we have. Figure 4.1 shows an image matrix including several basic image properties such as the RGB, HSV, and grayscale histograms. Each channel in different colour spaces are displayed separately, making it more obvious to observe the distinctive features. For example, foreground features could be more recognizable in grayscale histogram than in RGB histograms, or in some other cases, certain colour could be more easily observed in HSV colour space than in RGB colour space.

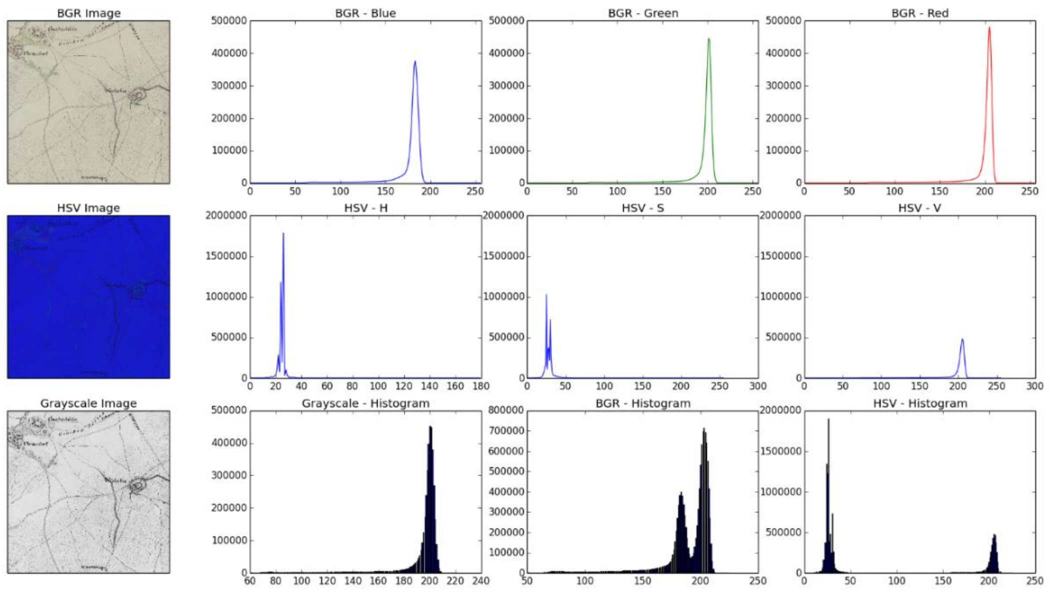


Figure 4.1 Image property matrix of a test map.

There are lots of choices of modules and libraries for producing these image properties matrices. In this case one called Matplotlib was applied to plot out both the histograms and the matrix.

Matplotlib is a Python 2D plotting library, including histograms, various types of charts, or even 3D plots ("Matplotlib: Python plotting", 2017). Especially for plotting diagrams, there are adequate existing plotting commands, making it possible to customize every single feature in the plot according to the user's needs.

The properties of each test image were obtained through OpenCV. It is the Python library for the famous library dedicated to algorithms related to Computer Vision and Machine Learning ("Introduction to OpenCV-Python Tutorials", 2014). It provides new possibilities in the field of image processing with one of the most popular programming languages. Besides, OpenCV makes use of Numpy, which makes it easier to get started with and could better integrate with other libraries such as SciPy and Matplotlib (2014).

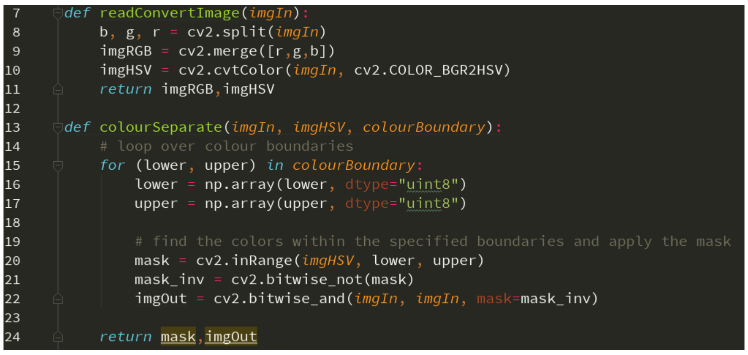
Here some basic functions are applied. The function cv2.imread() reads the image and provides access to the properties, such as image shape, data type, colour space. Then the images are displayed in Matplotlib instead of using the original OpenCV display window. This is due to a higher customizability in the Matplotlib interface. Almost every single feature of the plotting interface could be adjusted using corresponding commands, as well as choosing different colour maps for the colouring of the image and the diagrams. For each image, a 3*4 matrix was created, including the original image in RGB colour space, the image in HSV colour space and grayscale, and the corresponding histograms.

4.2 Colour Separation

Handling colour properly is essential in processing colour maps. For some of the test maps which include obvious large colour blocks, such as areal vegetation or hydrological features, using colour separation technique could separate them from the foreground effectively.

In order to take control of the colour value more easily, a colour space with improved perceptual uniformity is required (Salvatore & Guitton, 2004). Most of the scanned colour maps are in RGB colour space, which is the most prevalent colour space for displaying digital contacts. The reason for its popularity is that the RGB colour model which it uses works similar as the human visual system ("RGB color space - Wikipedia", 2017). However, according to Salvatore & Guitton, a main limitation in the uniformity of RGB leads to perceptual drawbacks, resulting in bins and holes in the colour space. Moreover, in most cases it is harder for a human then a machine to distinguish the colour only by the value of three chromaticities (red, green, and blue).

On the other hand, another proposed colour space, HSV, was used in the colour separation procedure. HSV is in represent of Hue, Saturation, and Value. As what its name indicates, the colour hue is controlled by the value of Hue. The primary and secondary colours (red, yellow, green, cyan, blue, and magenta) are arranged from 0° to 360°, making it more perceptually straightforward to create connections between colour shades and number of the value. In addition, the Saturation value stands for the saturation of the colour, and Value adjusts the brightness of the colour. In Python, the value ranges differently from the definition, although the colour hue still follows a spectral order. Hue ranges from 0 to 179, while Saturation and Value range from 0 to 255.



```python
def readConvertImage(imgIn):
    b, g, r = cv2.split(imgIn)
    imgRGB = cv2.merge([r,g,b])
    imgHSV = cv2.cvtColor(imgIn, cv2.COLOR_BGR2HSV)
    return imgRGB,imgHSV

def colourSeparate(imgIn, imgHSV, colourBoundary):
    # loop over colour boundaries
    for (lower, upper) in colourBoundary:
        lower = np.array(lower, dtype="uint8")
        upper = np.array(upper, dtype="uint8")

        # find the colors within the specified boundaries and apply the mask
        mask = cv2.inRange(imgHSV, lower, upper)
        mask_inv = cv2.bitwise_not(mask)
        imgOut = cv2.bitwise_and(imgIn, imgIn, mask=mask_inv)

    return mask,imgOut
```

Figure 4.2 Functions for colour space conversion and colour separation.

The function of converting the colour space to HSV and separating a certain colour are shown in Figure 4.2. The complete code could be found in ColourSeparation.py. The former one makes use of a colour converting function in OpenCV. In the latter one, both upper and lower boundaries are set for a certain colour hue. For instance, a range of [30, 110] was used for Hue in order to subtract the bluish green representing vegetations in the test map. Then, a mask should be formed containing the pixels which are in the colour range. At last, the pixels which are outside the given colour range are extracted with a logic operation between the input image and the mask. The result is shown in Figure 4.3.

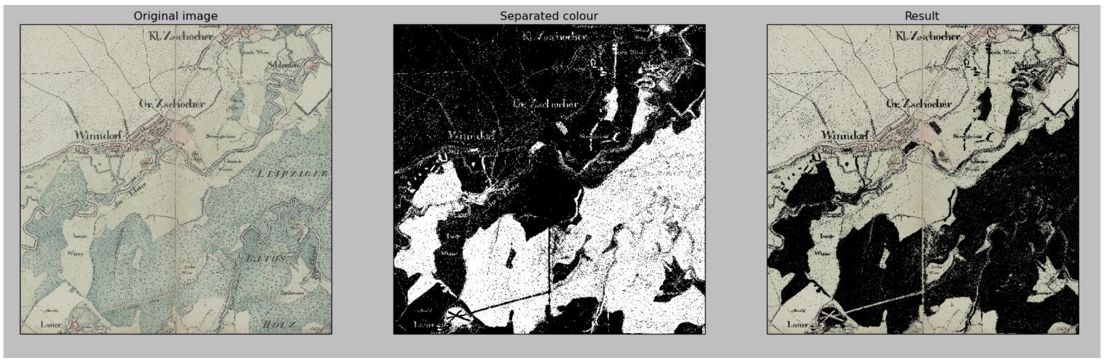


Figure 4.3 Colour separation results.

From what is shown in Figure 4.3, it should be obviously to observe that the green pixels are subtracted from the original image.

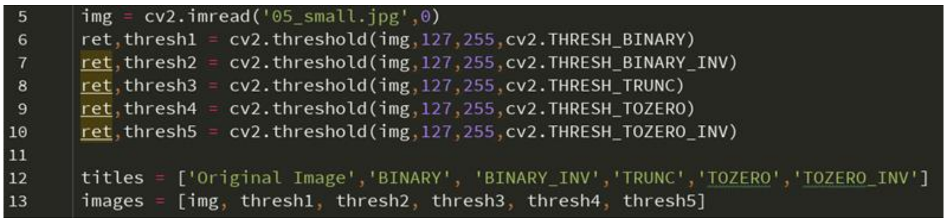
4.3 Binarization

After getting rid of the affection of extra distinctive colours, or if the image itself contain few special colours, the map image could now be read as a grayscale image and ready for pre-processing.

At pre-processing stages, grayscale images need to be binarized. There are plenty of modules available for image binarization, some of which were tested in this project:

- Thresholding in OpenCV

OpenCV provides a straight forward solution for image thresholding. There are two functions, cv2.threshold and cv2.adaptiveThreshold could be adapted. Both use a grayscale image as input, with a threshold value regarded as an indicator for classifying the pixels. Pixels in the input image will be classified and then assigned a new value depending on whether it is more (or sometimes less) then the threshold value. ("OpenCV: Image Thresholding", 2016). Here all five thresholding methods performed are global. Part of the code for image thresholding is shown in Figure 4.4 (Complete code in GlobalThreshold.py) a and the results could be reviewed in Figure 4.4 b.



```
img = cv2.imread('05_small.jpg',0)
ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)

titles = ['Original Image','BINARY', 'BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
```

Figure 4.4 a Thresholding using OpenCV functions.

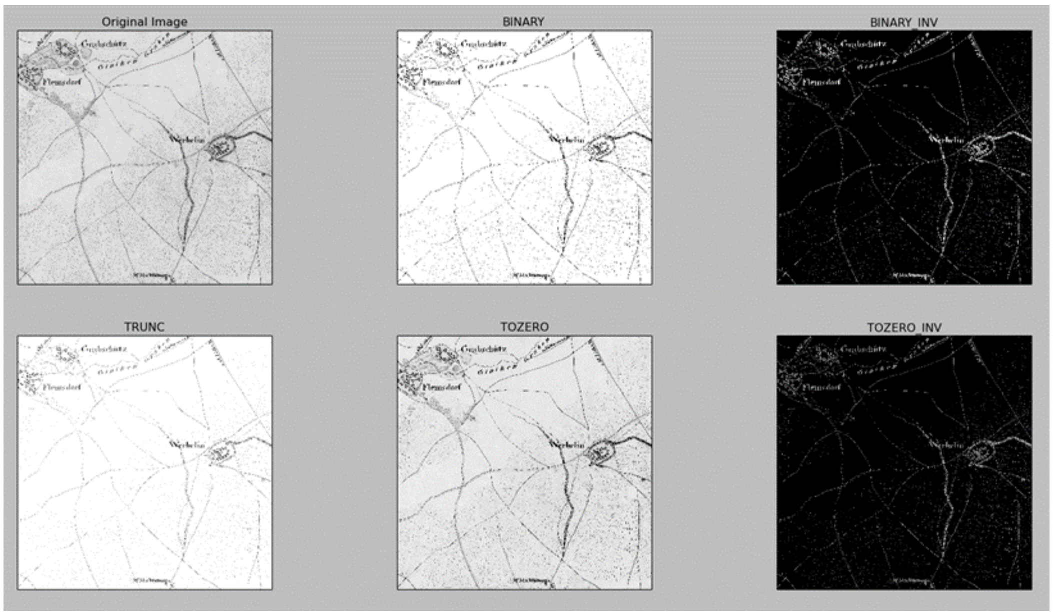
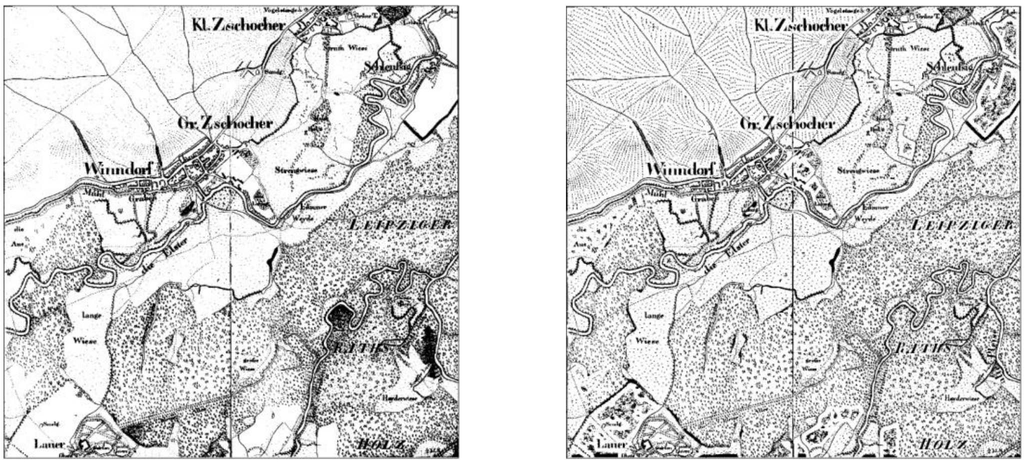


Figure 4.4 b Thresholding results.

In contrast to the global method, another one is adaptive thresholding. Instead of applying a single value to all the pixels in the input image, cv2.adaptiveThreshold breaks the image into small blocks and perform the thresholding calculation. Usage of adaptive thresholding is similar to the global one (Complete code could be found in AdaptiveThreshold.py). Figure 4.5 shows a contrast between the global thresholding and an adaptive one. As it could be clearly observed from the figure, compared to the global thresholding, when applying a same thresholding method, more details could be preserved through local thresholding. Correspondingly, more background information was remained after the local process. A main influence could be the clear enhancement of the hachures. Since the hachures should be removed as part of the background information, applying global threshold methods in this case could be more compatible.



a Global Otsu's thresholding b Local Otsu's thresholding

Figure 4.5 Comparison between global and local thresholding.

- Thresholding in ImageJ(Fiji)

ImageJ is an open source image processing interface based on Java. With plenty of plugins and packages, it is possible to solve different tasks by a few clicks ("ImageJ", 2017). Fiji is one of the distributions of ImageJ, with easy installation but powerful function. Using ImageJ could also implement either a global or an adaptive threshold function. Both of them could be applied via the adjust function under the image menu.

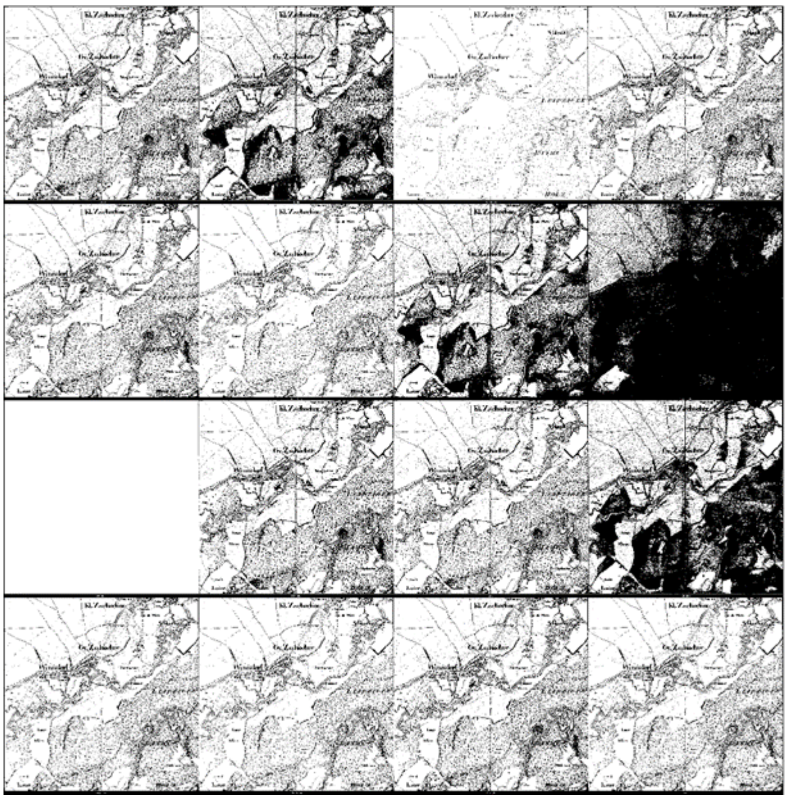
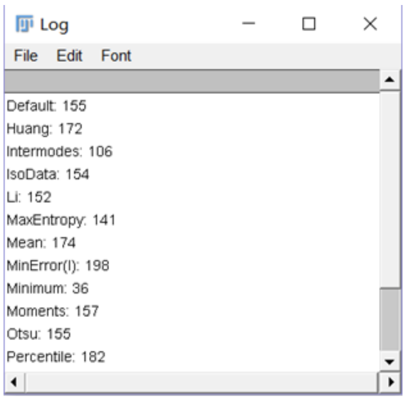


Figure 4.6 a Results of thresholding in Fiji.



```
Default: 155
Huang: 172
Intermodes: 106
IsoData: 154
Li: 152
MaxEntropy: 141
Mean: 174
MinError(I): 198
Minimum: 36
Moments: 157
Otsu: 155
Percentile: 182
```

Figure 4.6 b The thresholds in corresponding to the figure above.

The result matrix and the corresponding threshold value could be found in Figure 4.6. The result gives out a vivid comparison within different thresholding algorithms, from which the user could judge which one could be the most suitable for current input image. Here, two thresholding algorithms were applied for test maps.

Otsu's threshold: In situations when the histogram of the image appears to have an obvious bimodal property, Otsu's threshold could be applied. This algorithm tends to maximize the separability of the grayscale classes (Nobuyuki, 1979). In other words, an optimum value will be set as threshold if it could separate the two peeks in the histogram and reflect the intra-class variance as well. In Figure 4.6, the result matrix also includes a result from Otsu's algorithm.

In OpenCV, there is also existing method for this algorithm, namely cv2.THRESH_OTSU, which could be used as a parameter in the threshold function. Results of this method is displayed in Figure 4.7. From what is shown in the histogram, the bimodal property of this test map is not obvious, but still a minor peek could be found in area of lower grayscale. Thus, the outcome could still be predictable after thresholding. The blurred and dotted background becomes clearer, and the hachures are weakened as well. On the other side, darker pixels such as the edge lines and the characters are enhanced with an obvious improvement of intern pixel unity. This could be useful to later work, such as filtering and subtraction of clustered objectives (e.g. characters).

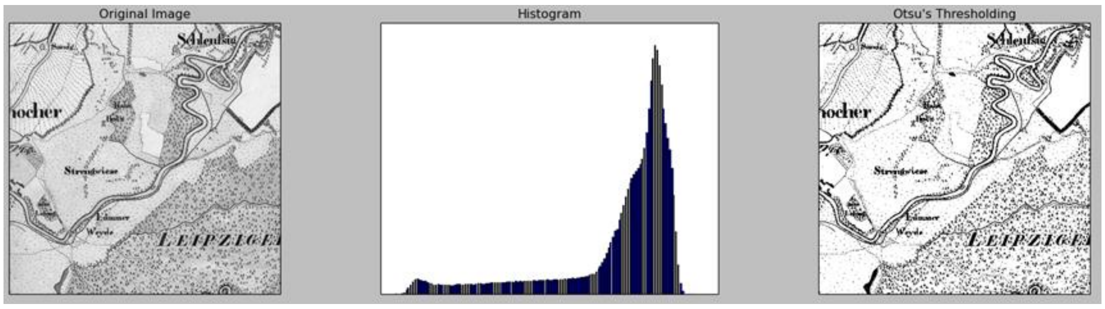


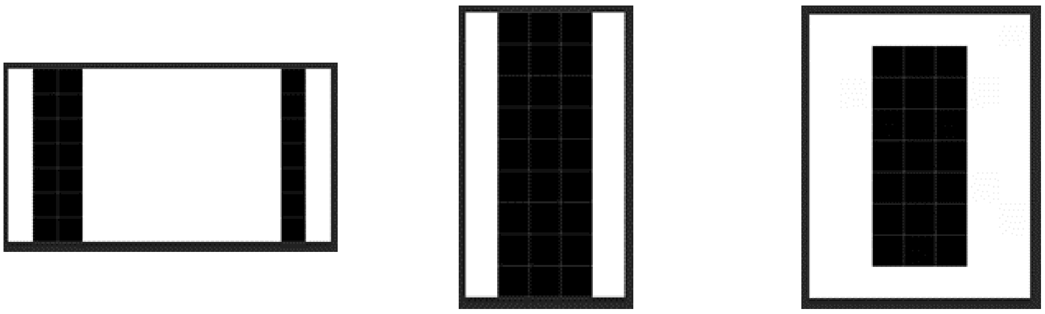
Figure 4.7 Result of Otsu's algorithm.

Triangle method: According to Zack G. W. et al., in the triangle algorithm, the threshold should be determined by normalizing the height and the dynamic range of the intensity histogram (1977). For some images that appears to have their maximum near one of the extreme points in the histogram.

4.4 Filtering

After the binarization stage, part of the background pixels has been removed. To detect the exact line features in the remaining foreground features, corresponding profile patterns need to be created and used for recognizing line features.

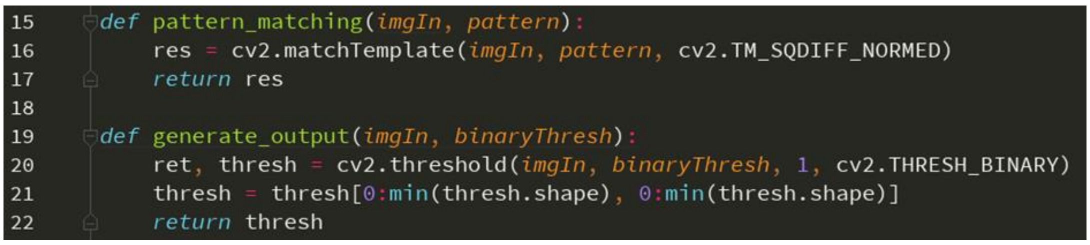
As shown in Figure 4.8, there are in total three pattern types of the line features are concluded. Each pattern is taken from the binarized image and then pruned, later saved as an individual pattern image clip. The width and length of the pattern are controlled as odd numbers, in order to set the centre of the pattern within exact one pixel. Figure 4.8 a represents the double-line roads in the test map, with one thicker line and one thinner line. Figure 4.8 b is the pattern for a single line road feature, while Figure 4.8 c is used to recognize the dash lines in the map image.



a Type 1 b Type 2 c Type 3

Figure 4.8 Profile pattern of three road types.

Then the patterns could be used for making a pattern matching. The test image will be traced by each pattern in four directions, horizontal, vertical, and both diagonal. There are already built-in functions in OpenCV for performing pattern matching, namely cv2.matchTemplate. This function provides access to searching and finding the template in a larger image, with a simple action to slide the template over the input image and to trace pixel after pixel (“OpenCV: Template Matching”, 2016). Similar to thresholding, template matching in OpenCV also provides several methods to be chosen from. In addition, the output image size would become smaller than the input one. Both length and width will be subtracted by the difference between the size of the input image and the pattern, and then plus one. Because of this difference, we need to cut the output image and take the minimum of both the length and the width. Figure 4.9 shows the functions for pattern matching and generating output (complete code in PatternMatch.py).

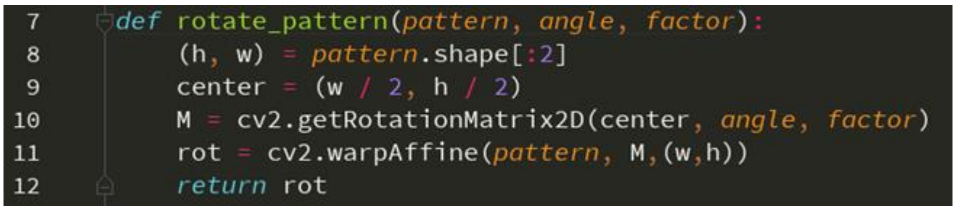


```python
def pattern_matching(imgIn, pattern):
    res = cv2.matchTemplate(imgIn, pattern, cv2.TM_SQDIFF_NORMED)
    return res

def generate_output(imgIn, binaryThresh):
    ret, thresh = cv2.threshold(imgIn, binaryThresh, 1, cv2.THRESH_BINARY)
    thresh = thresh[0:min(thresh.shape), 0:min(thresh.shape)]
    return thresh
```

Figure 4.9 Part of the functions during pattern matching.

The pre-generated pattern images for each type of roads include both horizontal and vertical direction. As it is displayed in Figure 4.10, for two diagonal directions, a simple rotating function was performed with the help of some other basic OpenCV functions.



```python
def rotate_pattern(pattern, angle, factor):
    (h, w) = pattern.shape[:2]
    center = (w / 2, h / 2)
    M = cv2.getRotationMatrix2D(center, angle, factor)
    rot = cv2.warpAffine(pattern, M,(w,h))
    return rot
```

Figure 4.10 The rotation function.

The results of individual output images could be found in Figure 4.11 a, in which a double-line pattern was used to trace in the four directions mentioned above. Reliable results show that the output image of each direction appears to obtain a strong tendency to the corresponding direction. Figure 4.11 b gives out the result when overlaying together all the output images of all types of pattern matching. Overlaying could be completed through the built-in logic operations in OpenCV (cv2.bitwise_and()).

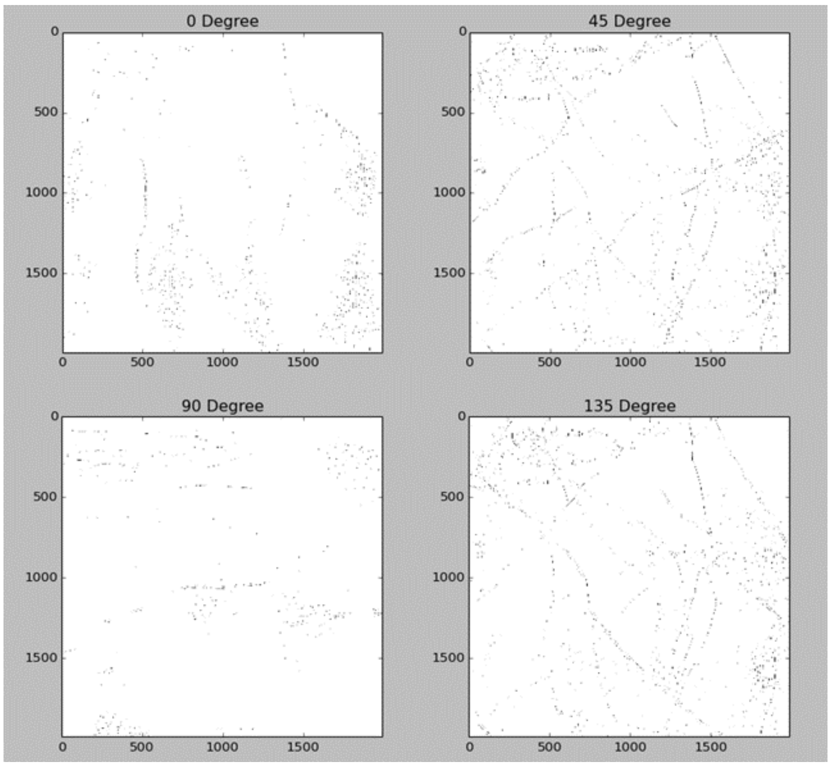


Figure 4.11 a Individual results of pattern matching.

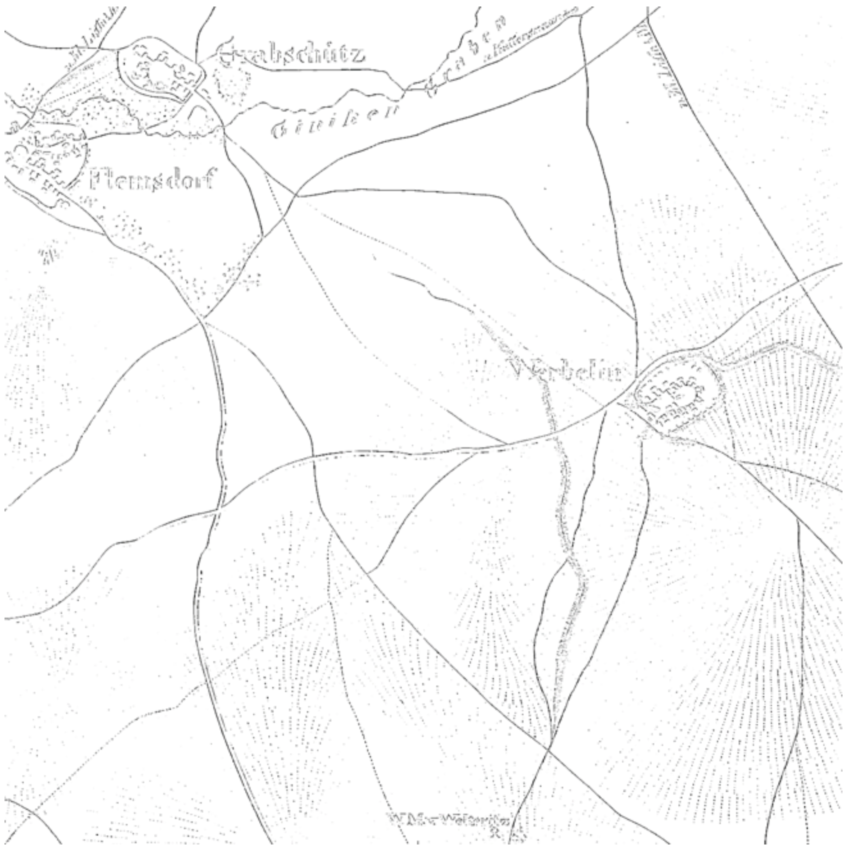


Figure 4.11 b Overlaid result of pattern matching.

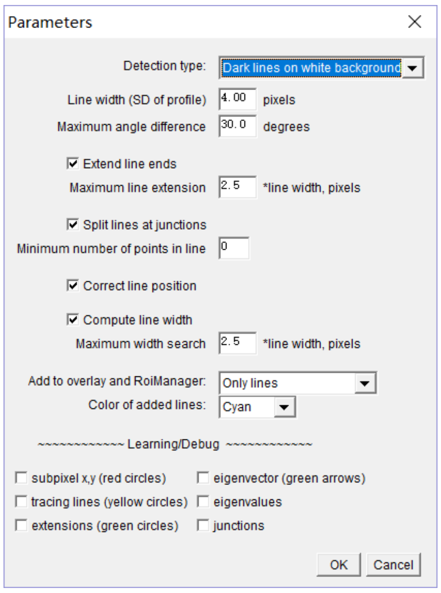
At the end of this stage, a pruned foreground image could be generated, containing necessary features but also distractors. It is worth to keep in mind that it is normal for pattern matching to extract some noise pixels, since the length of the patterns must be short enough to detect small curves in the line features. Some of these noises could be filtered in later procedures.

4.5 Line Feature Identification

Now that the foreground features are already separated from the background, and the lines are filtered according to different road type patterns, however, there are still noises and unnecessary features, nor the line features are well identified. As the filtering result shows, the unnecessary features include edge of the characters, small clusters of buildings in residential area, and the hachure lines that are in the same directions which are used for sliding the patterns.
A plugin called "Curve Tracing" could be found in ImageJ, which could provide a straightforward function to extract line features from the input image ("Curve Extraction Plugin", 2015). It follows one of the basic but effective algorithm proposed by Steger (1996).

This plugin works as follows:

The width and difference in angel are basic input parameters, being accompanied with optional choices of the solutions to line ends and intersections. The settings are shown in Figure 4.12.



```
Parameters                                                    X

                 Detection type:  Dark lines on white backgrounc
      Line width (SD of profile)  4.00   pixels
        Maximum angle difference  30.0   degrees

              ☑ Extend line ends
         Maximum line extension   2.5    *line width, pixels

              ☑ Split lines at junctions
Minimum number of points in line  0

              ☑ Correct line position

              ☑ Compute line width
            Maximum width search  2.5    *line width, pixels

   Add to overlay and RoiManager: Only lines
           Color of added lines:  Cyan

     ~~~~~~~~~~~~~ Learning/Debug ~~~~~~~~~~~~~

☐ subpixel x,y (red circles)      ☐ eigenvector (green arrows)
☐ tracing lines (yellow circles)  ☐ eigenvalues
☐ extensions (green circles)      ☐ junctions

                                               OK   Cancel
```

Figure 4.12 Parameter setting in Curve Tracing.

Before starting the extraction procedure, the plugin should return with a colourmap and threshold settings are required. The colourmap is shown in Figure 4.13 a. Blue lines in the image indicate the pixels that will be traced when the program runs, while the green part shows exactly where the tracing will start ("Curve Extraction Plugin", 2015). These two parameters are intuitively controlled by scroll bars in separated window (as shown in Figure 4.13 b).

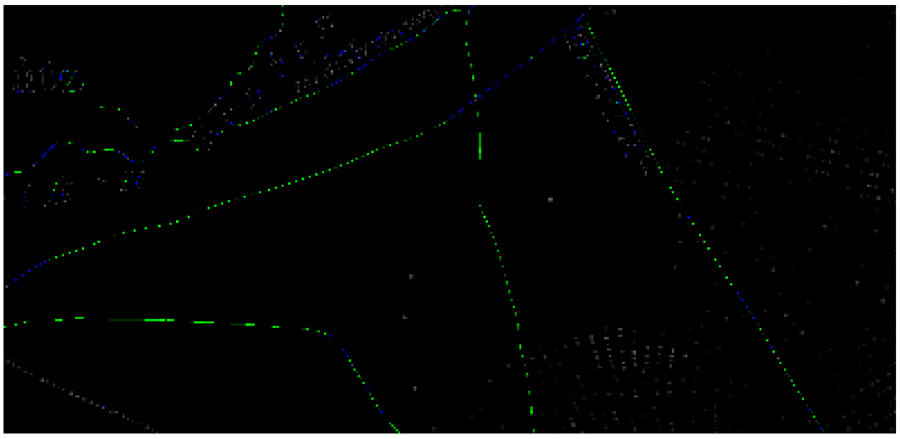


Figure 4.13 a Colourmap in Curve Tracing.

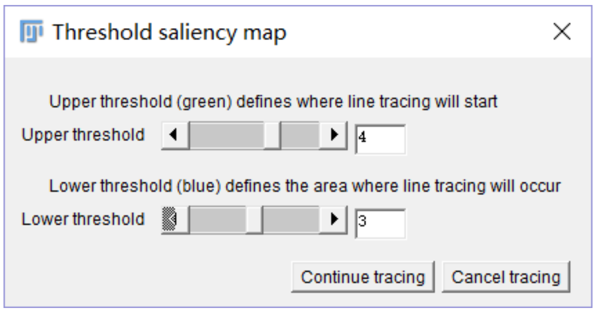

Threshold saliency map
Upper threshold (green) defines where line tracing will start
Upper threshold
4
Lower threshold (blue) defines the area where line tracing will occur
Lower threshold
3
Continue tracing
Cancel tracing

Figure 4.13 b Thresholding in Curve Tracing.

The result of curve tracing is displayed as an overlay cyan (or other colours) line layer onto the original input image. A clip of the image is shown as Figure 4.14. It returns a satisfied result for tracing the pre-recognized three road types, as well as some meandering small curves in the image. At the meantime, it is also clear that the method avoided the distinct dash lines and small broken pixel clusters. This could be a double-edged sword, meaning that the method could skip the hachure lines and the dash line features at the same time, which will be discussed in the later part.

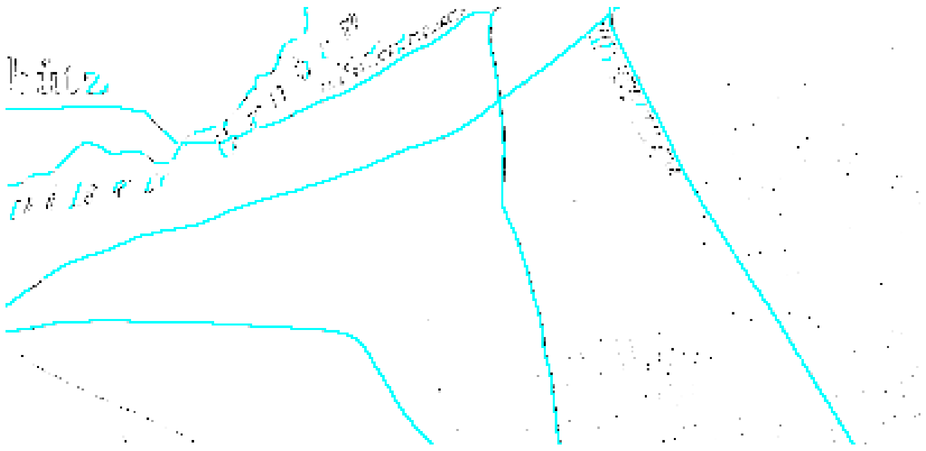


Figure 4.14 Traced curved lines in Fiji.

The traced lines were saved in ROIs, which could be examined in the ROI manager. Then the coloured trace lines could be again flattened into the input image, in order to extract only the traced lines. Similarly, the extraction of traced lines could take use of the colour separation method, since image is composed of highlighted lines and a binarized background. The method returns with white lines on a black background. This is performed separately for different types of line, below in Figure 4.15 shows a comparison between the traced ROI lines, the flattened overlay image, and the extracted line features.

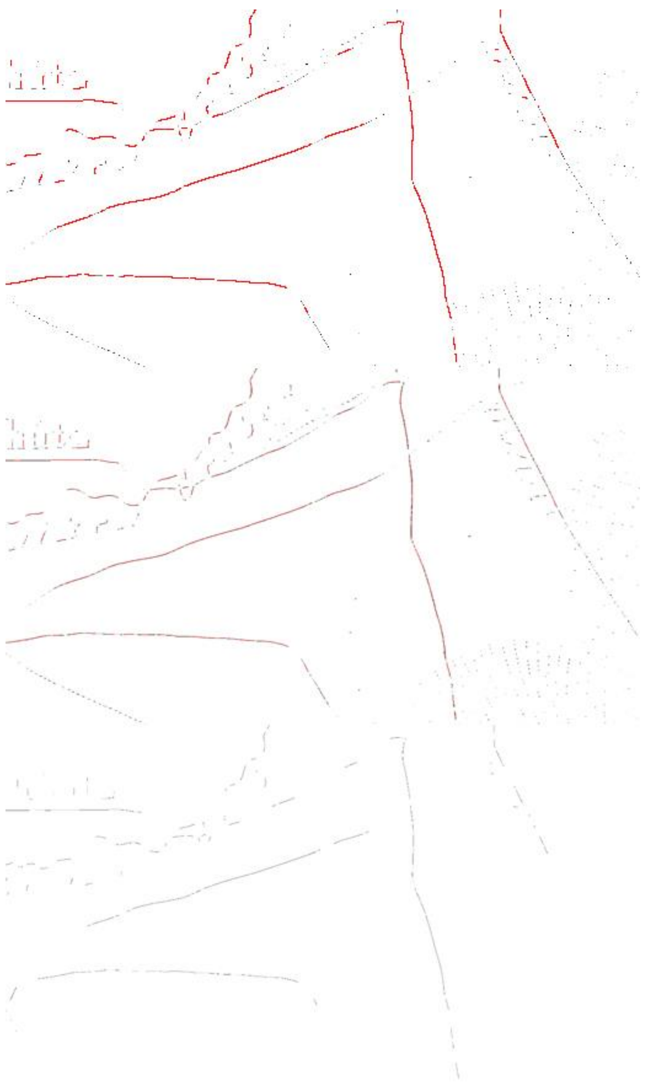
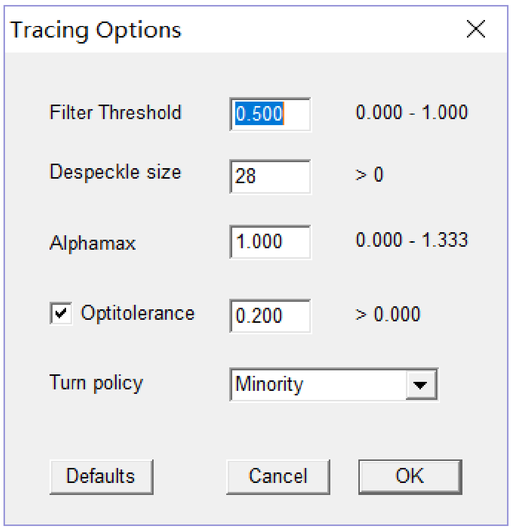


Figure 4.15 Compared results.

4.6 Vectorization

There are a few toolkits available for transforming raster images into vector file formats. Potrace, which is one of them, developed by Peter Selinger, is a quite well-developed project focusing on converting bitmaps into vector files ("Peter Selinger: Potrace", 2017). In addition to the project itself, a considerable number of software, interfaces, and services are built based on Potrace, including both free and commercial ones. Here in this paper, one called CR8tracer is applied in the vectorization stage. It is a graphical user interface based on Potrace by Allan Murray ("CR8 Software Solution", 2016). It takes bitmaps as input images and could export results in PS, EPS, SVG, or GFS vector formats. Some of the formats are also compatible with other font editing software. Tracing options are corresponded to those in Potrace. As shown in Figure 4.16, filter threshold decides the lowest gray value that will be converted. Despeckle size sets the largest size of the noises to be removed. Alphamax controls the threshold of the corners, with a default value of 1. Optitolerance is the tolerance for curve optimization, the default value of which is 0.2. Both tracing and type settings are available in the menu. While in vectorization stage, only tracing options are considered. Because the characters are already broken during in former stages and could not be recognized as fonts.


Tracing Options

Filter Threshold	0.500	0.000 - 1.000
Despeckle size	28	> 0
Alphamax	1.000	0.000 - 1.333
[✓] Optitolerance	0.200	> 0.000
Turn policy	Minority	

Figure 4.16 Tracing settings.

The result of an output SVG is shown in Figure 4.17. Noises could be further removed by applying a despeckle size. However, due to the continuity in the extracted line features, part of the necessary line features will also be removed if the size value is too small.

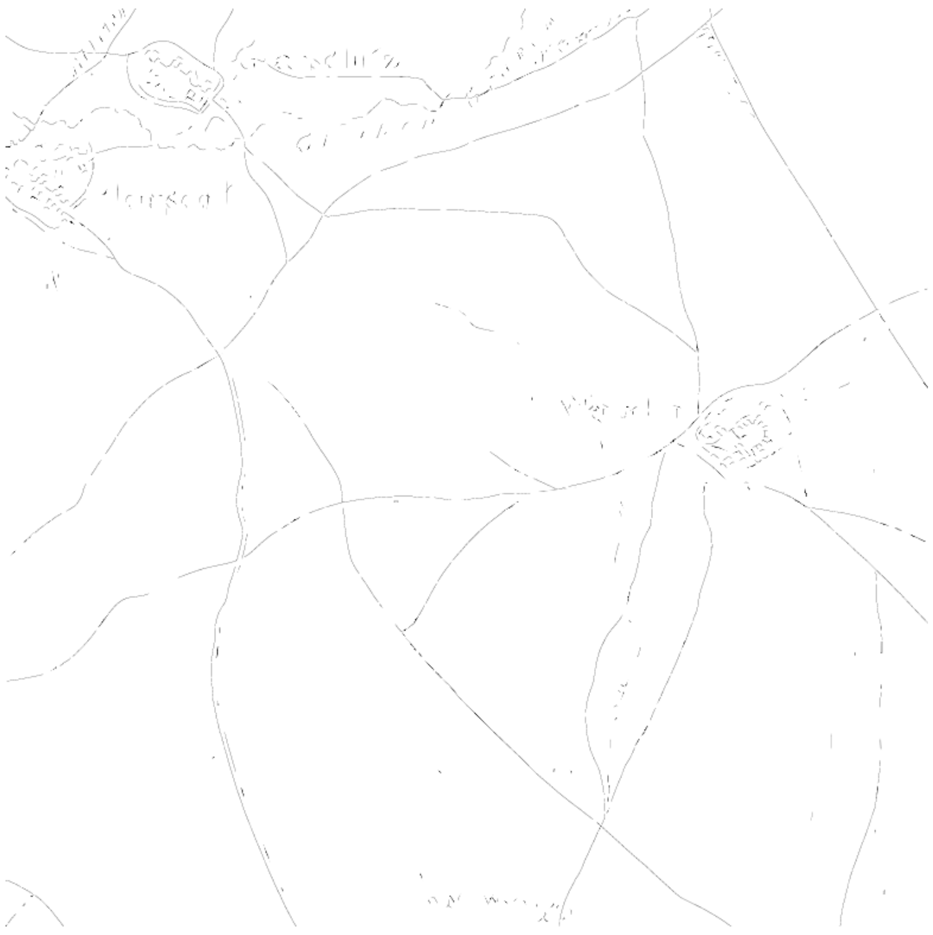
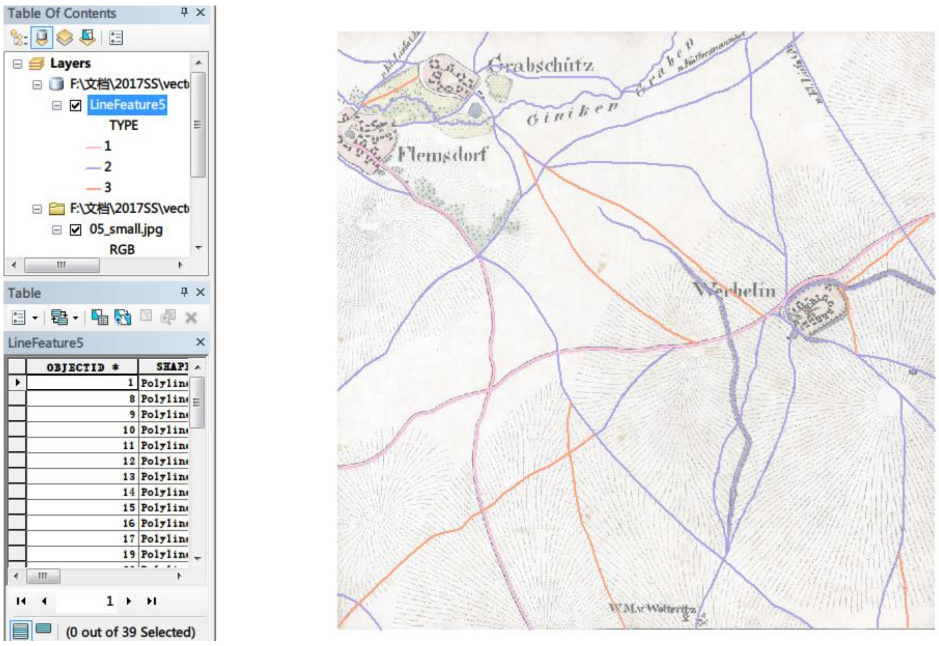


Figure 4.17 SVG output.

4.7 Building Referencing System

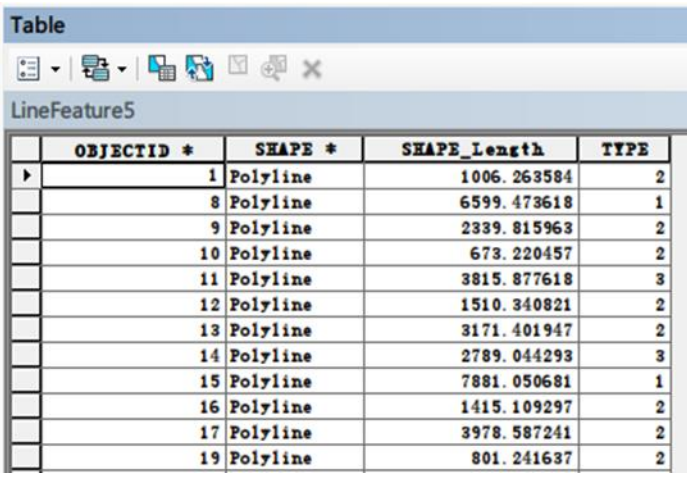
It is always notable to pay special attention to the assessment of the overall quality of the vectorization results, for which purpose, a referencing system will be needed after the extraction work.

Since the original map data are all in image format, extra work for building the raster references are needed. The manual vectorization of the line features has been done in GIS software. Vector features were saved in line feature classes in geodatabases. Figure 4.18 a displays the line feature added onto the original map and b represents the attribute table of the vectorized feature class, with four fields indicating necessary properties of the features, namely OBJECTID, SHAPE, SHAPE_Length, and TYPE. All the fields were generated automatically, except the last one, representing assorted types of the line features judged by the line type on the map.



OBJECTID *	SHAP
1	Polyline
8	Polyline
9	Polyline
10	Polyline
11	Polyline
12	Polyline
13	Polyline
14	Polyline
15	Polyline
16	Polyline
17	Polyline
19	Polyline

Figure 4.18 a The vectorized line features in ArcGIS.



Table

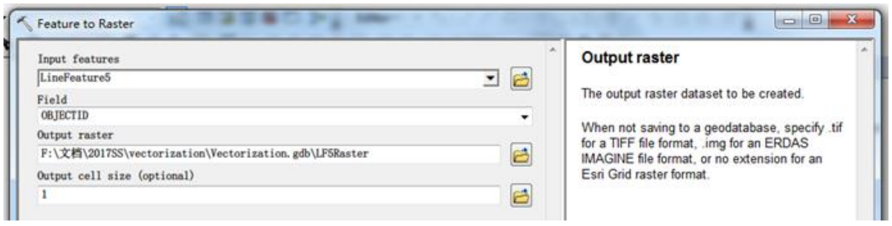
LineFeature5

OBJECTID *	SHAPE *	SHAPE_Length	TYPE
1	Polyline	1006.263584	2
8	Polyline	6599.473618	1
9	Polyline	2339.815963	2
10	Polyline	673.220457	2
11	Polyline	3815.877618	3
12	Polyline	1510.340821	2
13	Polyline	3171.401947	2
14	Polyline	2789.044293	3
15	Polyline	7881.050681	1
16	Polyline	1415.109297	2
17	Polyline	3978.587241	2
19	Polyline	801.241637	2

Figure 4.18 b The attribute table of the line features.

In order to make comparison between the extracted data and the pre-vectorized data, a format transformation need to be carried out for both sides. For the pre-vectorized line feature, a built-in module called "Feature to Raster" in ArcGIS could be taken use of ("Feature to Raster - Conversion toolbox", 2017). Point, line, and polygon features could be converted into raster datasets by using this module (Figure 4.19 a).

The setting of cell size of the output raster dataset requires extra attention. The cell size option is in control of the resolution of the output raster dataset. In order to keep the same resolution as the original map file, the cell size should be the same instead of using default value. By default, the cell size is always set as the shortest of the width or the height of the input feature divided by 250 in the output spatial reference. In this case, the cell size need to be modified through changing the settings in the environment, which is displayed in Figure 4.19 b. The cell size of the original map file could be taken from the “Snap Raster” option, and the conversion result could be found in Figure 4.20.


Feature to Raster

Input features
LineFeature5

Field
OBJECTID

Output raster
F:\文档\2017SS\vectorization\Vectorization.gdb\LF5Raster

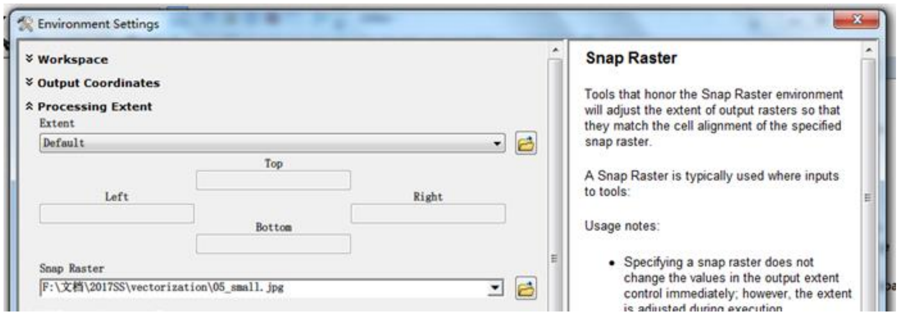
Output cell size (optional)
1

Output raster

The output raster dataset to be created.

When not saving to a geodatabase, specify .tif for a TIFF file format, .img for an ERDAS IMAGINE file format, or no extension for an Esri Grid raster format.

Figure 4.19 a Conversion from feature to raster dataset.


Environment Settings

Workspace
Output Coordinates
Processing Extent
Extent
Default
Top
Left
Right
Bottom

Snap Raster
F:\文档\2017SS\vectorization\05_small.jpg

Snap Raster

Tools that honor the Snap Raster environment will adjust the extent of output rasters so that they match the cell alignment of the specified snap raster.

A Snap Raster is typically used where inputs to tools:

Usage notes:

- Specifying a snap raster does not change the values in the output extent control immediately; however, the extent is adjusted during execution

Figure 4.19 b Adjustment on cell size.

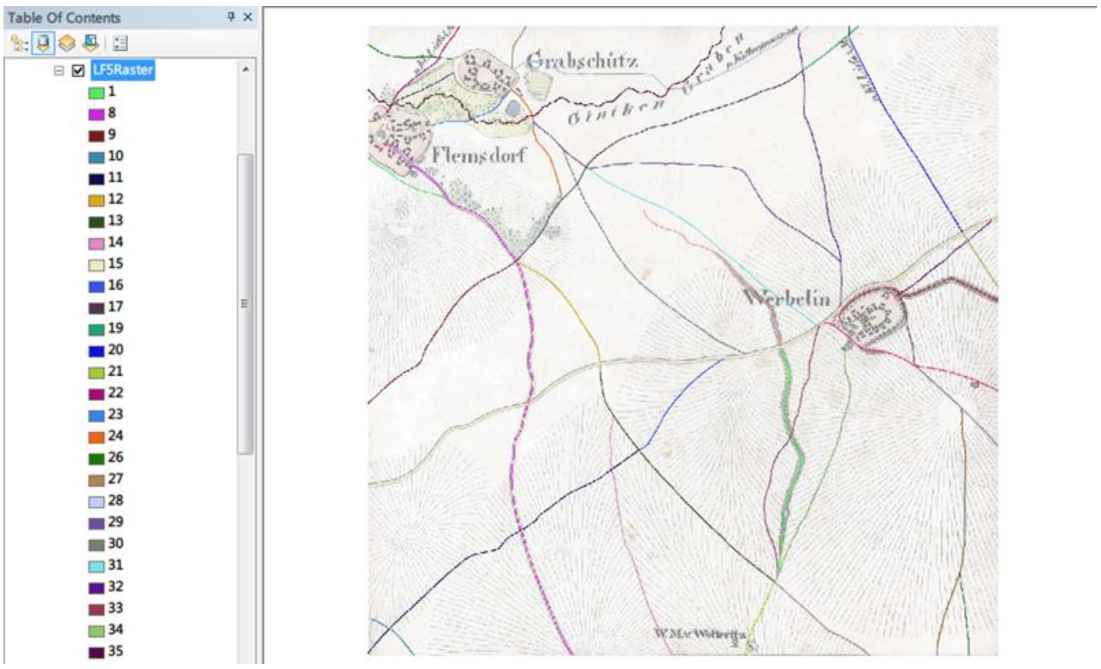


Figure 4.20 The converted raster dataset for referencing.

In order to investigate the effect of different pattern matching, individual images are generated according to different types of line feature. The process of converting an SVG file into PNG format could be completed with some extra python packages. One of which is called Wand, an ImageMagick binding for Python ("Wand – Wand 0.4.4", 2016). With some lines of simple code, it could complete the task of converting the input vector file into the assigned image format. Unfortunately, the script could only work in either an earlier version of Python 2.7 or in Python 3. The LocalLibrary argument always met some trouble when working under Python 2.7 as it only accepts string instead of Unicode.

Another attempt is CairoSVG, which is also reported to be a well-functioned Python package for conversion ("Cairosvg", 2017). However, only Python 3 is supported and the Python 2 support has already been dropped.

So here in this paper, PS format was used instead of SVG format to save the vectorized data. It could be processed in image processing software and then easily converted to PNG format. Now that we have both versions of the PNG images. The referencing procedure is as well quite straightforward. Another logical operation is performed to those two images. The principle is that to add those two images together and make comparison to each pixel. Since the two images are all binary image, the comparison also acts as a hit-or-miss procedure. If the pixel position in both images are filled with black colour, then this pixel would be marked as a "blank" one, indicating the line feature in both images are matched. On the other hand, if the pixel remains black, it means that the line feature two images failed to match with each other. The reason will be discussed in the last chapter. Figure 4.21 shows a part of the result image representing a single line type in the test map. The colour map is adjusted for a better visual effect.

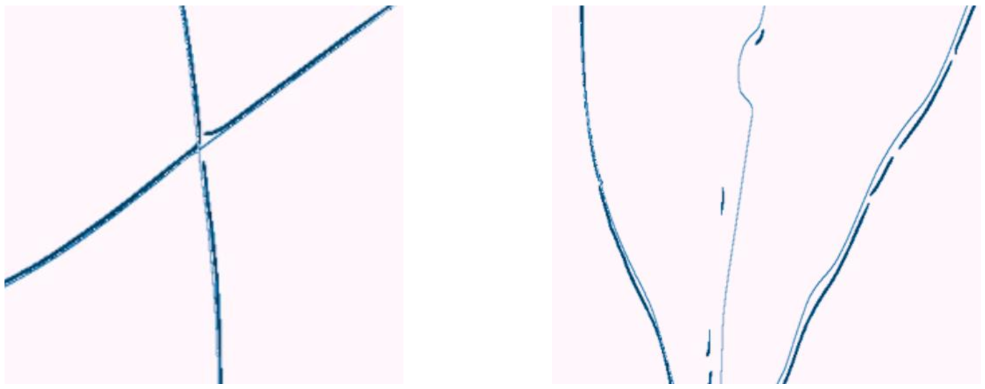


Figure 4.21 Result of a reference for line type 2.

5 Conclusion

In the last chapter, a complete workflow was discribed from accessing the image properties to the final logical operations. The result of the last opeartion returned with some facts that worth noting:

First, the displacement between the two referencing images, to some extent, largely affected the logical operation result. Although both of the images are in a same size, the image content have experienced some slight shifting, the reson of which might be the pattern matching stage. When a pattern is applied and scanned through the input image, the size of the result image would be reduced. Besides, as it could be observed from Figure 4.21 a, obviously, it should be the same line being shown on both images, yet, the pixels couldn't totally coincide with each other. Part of the pixels are subtracted, resulting in blank areas somewhere between the lines. Therefore the what the output image shows could be deviated from the actual results. However, the paradox is that changing cell size when using the "Feature to Raster" tool could slightly increase the performance, but it will greatly decrease the resolution as well. Thus, somehow a balance should be maintained, where the overlaying method could achieve the best effect and at the same time, the resolution shold be maintained within an acceptable torelance. Moreover, from what Figure 4.21 b reflected, the line in the middle suffers from worse extraction results compared to the other two lines in the screenshot. If we reflect back to the original map image, or the vectorized dataset, it is not hard to find out that the line type is dictinct from the other two, which is a dashed line with dense small pixel clusters surrounding the edge of the line (Type 3). Taking the colour properties of historical maps into consideration, simply performing a normal colour separation could be hard to reach a satisfied result. Therefore, using patterns for filtering was hoped to export reliable results. Meanwhile, extraction of this type of line features is much challenging than the other two types, as the line shape itself resemble hachure lines very much. In some area of the image, the dots of dash lines could be smaller than those of hachure lines, for instance foot paths in suburbs and forest area. Besides, in most historical maps which are hand-painted, pixel size of each dot also vary a lot. These were the main obstacle during the filtering work. Although later the adoption of curve tracing algorithm could ease the problem a little bit, the result of applying a dash line pattern individually is still quite far from satisfaction.

Further work after this paper include finding a solution to the paradox between proper cell size and resolution, improving the methods of pattern matching and noise filtering. In addition, it is also worth to keep surveying for more useful open source tools, which could be either complex or handy. Many a little makes a mickle, and maybe new thoughts would enlightened during this process.

References

Cairosvg. (2017). Retrieved from http://cairosvg.org/

Cao, R., & Tan, C. L. (2001). Text/graphics separation in maps. *In International Workshop on Graphics Recognition* (pp. 167-177). Springer Berlin Heidelberg.

Chiang, Y. Y., Knoblock, C. A., & Chen, C. C. (2005). Automatic extraction of road intersections from raster maps. *In Proceedings of the 13th annual ACM international workshop on Geographic information systems* (pp. 267-276). ACM.

Chiang, Y. Y. & Knoblock, C. A. (2009). A method for automatically extracting road layers from raster maps. *In Document Analysis and Recognition,* 2009. ICDAR'09. 10th International Conference on (pp. 838-842). IEEE.

CR8 Software Solution. (2016). Retrieved from http://www.cr8software.net/tracer.html

Curve Extraction Plugin. (2015). Retrieved from http://katpyxa.info/feedbacks/?p=154

Dhar, D. B., & Chanda, B. (2006). Extraction and recognition of geographical features from paper maps. *International Journal of Document Analysis and Recognition (IJDAR)*, 8(4), 232-245.

Feature to Raster - Conversion toolbox. (2017). Retrieved from http://pro.arcgis.com/en/pro-app/tool-reference/conversion/feature-to-raster.htm

ImageJ - ImageJ. (2017). Retrieved from http://imagej.net/ImageJ

Introduction to OpenCV-Python Tutorials. (2014). Retrieved from http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_setup/py_intro/py_intro.html#intro

Khotanzad, A., & Zink, E. (2003). Contour line and geographic feature extraction from USGS color topographical paper maps. *IEEE transactions on pattern analysis and machine intelligence*, 25(1), 18-31.

Matplotlib: Python plotting. (2017). Retrieved from https://matplotlib.org/

Montanari, U. (1969). Continuous skeletons from digitized images. *Journal of the ACM (JACM),* 16(4), 534-549.

OpenCV: Image Thresholding. (2016). Retrieved from http://docs.opencv.org/3.2.0/d7/d4d/tutorial_py_thresholding.html

OpenCV: Template Matching. (2016). Retrieved from http://docs.opencv.org/3.2.0/d4/dc6/tutorial_py_template_matching.html

Otsu, N. (1979). A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics,* 9(1), 62-66.

Peter Selinger: Potrace. (2017). Retrieved from http://potrace.sourceforge.net/#description

Pouderoux, J., Gonzato, J. C., Pereira, A., & Guitton, P. (2007). Toponym recognition in scanned color topographic maps. *In Document Analysis and Recognition,* 2007. ICDAR 2007. Ninth International Conference on (Vol. 1, pp. 531-535). IEEE.

RGB color space - Wikipedia. (2017). Retrieved from https://en.wikipedia.org/wiki/RGB_color_space

Salvatore, S., & Guitton, P. (2004). Contour line recognition from scanned topographic maps. *In Proceedings of the Winter School of Computer Graphics* (pp. 1-3).

SLUB Dresden: Maps. (2016). Retrieved from https://www.slub-dresden.de/en/collections/maps/

SLUB Dresden: The Map Collection. (2016). Retrieved from https://www.slub-dresden.de/en/collections/maps/the-map-collection/

Steger, C. (1996, August). Extraction of curved lines from images. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on* (Vol. 2, pp. 251-255). IEEE.

Wand – Wand 0.4.4. (2016). Retrieved from http://docs.wand-py.org/en/0.4.4/

Wenyin, L., & Dori, D. (1999). From raster to vectors: extracting visual information from line drawings. *Pattern Analysis & Applications,* 2(1), 10-21.

Zack, G. W., Rogers, W. E., & Latt, S. A. (1977). Automatic measurement of sister chromatid exchange frequency. *Journal of Histochemistry & Cytochemistr*y, 25(7), 741-753.