Vesa Ala-Mattila

Alpha Logos Software Oy

August 31, 2023

Contents

1 General technical remarks

We start with some technical remarks regarding directories, non-standard code libraries, and conventions pertaining to images, coordinate systems and representation of geometric objects.

The root directory for example images is `./example_images`. When referring to example images of a specific type in this exposition, we will mention the appropriate subdirectory of `./example_images`.

The root directories for sample logbook data and sample logbook results are `./sample_logbook_data` and `./sample_logbook_results`, respectively. It is possible that `./sample_logbook_results` is empty, since the relevant directories and files are created by the main testing function if necessary.

The two non-standard Python libraries used in the code are `numpy` (numpy.org) and `cv2` (opencv.org) imported as `np` and `cv`, respectively. We will not give detailed references to the documentations of these libraries because said documentations have very adequate search functions.

The `numpy` version is 1.24.2 and the `cv2` version is 4.7.0. It is likely that the exact versions used are not critical – they simply have to be recent enough.

The color channel convention of `cv2` is BGR and not RGB!

Typically, the grayscale versions of the input images are used in the technical algorithms. Color images are used to describe the results.

Recall that a matrix (e.g. a `numpy` array) uses the row-column type coordinate system. That is, the value on row `row` and in column `column` is referred to by using an expression of the form `matrix[row, column]`.

Since the ordering of matrix rows is vertical and the ordering of matrix columns is horizontal, we use the symbol `y` to refer to rows and `x` to refer to columns. Therefore, it is natural to write an expression of the form `matrix[row, column]` as `matrix[y, x]`.

It is important to remember that many functions in `cv2` (e.g. functions which draw geometric objects such as line segments and rectangles) use the standard coordinate convention, i.e., a point with x-coordinate `x` and y-coordinate `y` is referred to by using the ordered pair `[x, y]`.

Therefore, the `numpy` expression `matrix[y, x]` refers to a point whose `cv2` representation is `[x, y]` – the order of the coordinate axes is switched!

A line segment is represented as `[[x_1, y_1], [x_2, y_2]]`, i.e., as an ordered pair of the endpoints. Typically, there is no specific meaning associated with the order, so `[[x_2, y_2], [x_1, y_1]]` would work just as well. The term 'line' is used to refer to line segments in the code.

The representation of a rectangle is also `[[x_1, y_1], [x_2, y_2]]`. The point `[x_1, y_1]` is the top-left corner point, and `[x_2, y_2]` is the bottom-right corner point.

The above representation of a rectangle implies that each side of the rectangle is parallel to either the x-axis or the y-axis, i.e., that the rectangle is xy-aligned. Indeed, unless stated otherwise, we assume that any rectangle considered in this text or in the code is xy-aligned.

2 The main algorithm

The file `main_computer_vision_functions.py` contains the code of the main algorithm. In addition to some file-specific global variables, the file contains the following functions:

```
detect_table_structure_and_elements
detect_table_structure
detect_horizontal_or_vertical_table_lines
detect_table_elements
```

In the following subsections, we will take a closer look at the global variables and each of the functions. The auxiliary functions used by these functions will be considered in the next section. (These auxiliary functions are contained in other files.)

2.1 Global variables

In this subsection, we take an initial look at the global variables. Their roles and properties will be clarified further when we discuss the actual algorithms in greater detail.

LSD_LINE_DETECTOR_SCALE: The library cv2 provides a few methods for detecting line-like structures in pictures. After testing, we ended up using the method based on the class LSDDetector, and LSD_LINE_DETECTOR_SCALE is a technical argument used by LSDDetector. A detailed technical discussion of LSDDetector is beyond the scope of this exposition. The standard value 2 given to LSD_LINE_DETECTOR_SCALE seems to work well, and we did no testing using other values.

LSD_LINES_IMAGE_COLOR and LSD_LINES_IMAGE_THICKNESS: These variables are used when the line-like pixels detected by LSDDetector are drawn in a zero-initialized image of the same shape as the input image, see progress images image_3.jpg and image_14.jpg in example_progress_images. The resulting image is supposed to be a binary image, and therefore the maximal value of 255 is appropriate for LSD_LINES_IMAGE_COLOR. The value of LSD_LINES_IMAGE_THICKNESS determines the connected components of the resulting image. If the value is small, say 1, line segments close to one another will not connect, so the value needs to be larger in order to improve connectivity. However, it is likely that the connectivity can be improved by only a limited amount through adjusting this value, although some testing using larger values might be a good idea. In any case, it is likely that this variable should be a function argument and not a global variable.

LSD_LINES_COMPONENT_RECTANGLES_IMAGE_COLOR and
LSD_LINES_COMPONENT_RECTANGLES_IMAGE_THICKNESS: These variables have their primary technical use when a number of collections of mutually-intersecting rectangles are drawn in a zero-initialized image of the same shape as the input image. They are also used in many progress images describing the main algorithm. See progress images image_6.jpg and image_17.jpg in example_progress_images for the technical usage. The main idea is that each of the collections contains exactly one of the relevant table lines and that each relevant table line is contained in one such collection. The maximal value 255 of LSD_LINES_COMPONENT_RECTANGLES_IMAGE_COLOR has the

justification as `LSD_LINES_IMAGE_COLOR` had above. We use the minimal value 1 for `LSD_LINES_COMPONENT_RECTANGLES_IMAGE_THICKNESS` because the primary geometric property of the rectangles in a collection is that they are mutually-intersecting (which we define to mean that any rectangle in a collection intersects at least one other rectangle in the collection). If the value is increased too much, it is possible that two collections accidentally connect or that some superfluous rectangles connect to some collection. In any case, it might be interesting to do some tests using larger values for the thickness parameter. Like `LSD_LINES_IMAGE_THICKNESS`, it is quite possible that `LSD_LINES_COMPONENT_RECTANGLES_IMAGE_THICKNESS` should be a function argument and not a global variable.

`LSD_LINES_FULL_IMAGE_COLOR` and `LSD_LINES_FULL_IMAGE_THICKNESS`: These variables are used in the construction of some progress images in a way which hopefully is rather self-explanatory. The BGR value $(255, 0, 0)$ corresponding to the color blue was chosen on a whim, and it is used multiple times throughout the code.

The rest of the global variables in `main_computer_vision_functions.py` serve similar purposes as the variables `LSD_LINES_FULL_IMAGE_COLOR` and `LSD_LINES_FULL_IMAGE_THICKNESS`, and so they will not be discussed in greater detail in this exposition.

2.2 Function *detect_table_structure_and_elements*

This is the main function in the file, and it is called by the functions used to test the main algorithm. There are two primary subtasks, namely the detection of the table structure, i.e., the relevant table lines, and the detection of table elements (basically everything meaningful except for the relevant table lines). In addition to the input image, the function receives a set of arguments pertaining to each of the primary subtasks. The structure of the function itself is hopefully rather self-explanatory due to the explicit argument and function names, and so the nature of the arguments and return values will be discussed in greater detail in the following subsections.

2.3 Function *detect_table_structure*

This function is used to detect the table structure in the input image, i.e., the relevant table lines. This function can also be used to construct so-called progress images (images which illustrate the functioning of the table line detection algorithm) and/or an image which displays the detected table lines.

In technical terms, the line-like structures in the input image are detected by the `detect` method of the instance `lsd_line_detector` of the `LSDDetector` class of `cv2`. The details as to the line detection algorithm implemented by `LSDDetector` will not be covered in this document. We mention only that `cv2` provides a number of line detection algorithms and, according to the tests so far, the one implemented by `LSDDetector` works the best.

The function argument `num_octaves` is a technical argument used by the `detect` method of `lsd_line_detector`. As in the case of the global variable `LSD_LINE_DETECTOR_SCALE`, we will not consider here in detail the technical nature of `num_octaves`. According to tests, setting the value of `num_octaves` to be 4 seems to work rather well (see `run_main_tests.py`). The `detect` method has also the `mask` argument which can be used to restrict the area processed by the method. In the current situation, we choose the maximal mask due to the large size of the logbook tables. Smaller masks presumably make good sense in other situations, and when those are considered, `mask` should be among the function arguments.

The return value of `detect` is a collection of objects called `lsd_lines`. Recall that a line segment (`line`) is represented by an ordered pair of two points. In terms of geometry, an `lsd_line` is also a line segment, but as an object data structure it is much more complex. For example, an `lsd_line` has the attribute `angle` which will be used when we filter line segments which are not close to being either horizontal or vertical. The file `lsd_line_functions.py` contains functions pertaining to `lsd_lines`, and we will consider this file in detail later in this document.

Once all of the relevant line segments (`lsd_lines`) have been detected, we use `detect_horizontal_or_vertical_table_lines` to construct collections of horizontal and vertical table lines. See the next subsection for a detailed discussion. We mention here that most of the function arguments, for example `sin_upper_bound` and `cos_upper_bound` on one hand, and `right_extra_length` and `bottom_extra_length` on the other, form pairs such that the first members of the pairs are used in the detection of horizontal table lines and the second members in the detection of vertical table lines.

We hope that the remaining details of `detect_table_structure` are self-explanatory given the above discussion and the explicit variable, argument and function names used in the code.

2.4 Function *detect_horizontal_or_vertical_table_lines*

Depending on whether the value of `detect_horizontal_lines` is `True` or `False`, this function detects the horizontal or vertical table lines in the input image, respectively. In the following, we will give a more detailed description of the algorithm steps.

We omit details pertaining to function arguments and variable names from this presentation. Please see the code itself for these details.

The directory `./example_images/example_progress_images` contains the progress image examples referred to in the description. We use the symbol `k` to refer to the image file `image_k.jpg`.

1) We use the function `filter_lsd_lines` (see `lsd_line_functions.py`) to filter `lsd_lines` which are not long enough or which are not sufficiently horizontal/vertical.

2) Draw the remaining `lsd_lines` in a zero-initialized image of the same shape as the input image. The relevant progress image examples for this step are `2`, `3`, `13` and `14`.

3) Using `compute_connected_component_parameters` contained in the file `general_computer_vision_functions.py`, determine the connected components in the image drawn in 2).

4) For each connected component determined in 3), we construct the minimal rectangle containing said component and extend the rectangle either right-

wards or downwards, depending on whether we are looking for horizontal or vertical table lines, respectively. The idea is that for each relevant table line there are `lsd_lines` close to it and, therefore, also connected components determined in 3) close to it. The hope is that, by extending the rectangles, each relevant table line will be contained in a collection of mutually-intersecting rectangles. (By definition, a collection of rectangles is said to be mutually-intersecting if for each rectangle in the collection there is at least one other rectangle in the collection which intersects the first rectangle). The relevant progress image examples for this step are `4`, `5`, `6`, `15`, `16` and `17`.

5) Draw the rectangles constructed in 4) in a zero-initialized image of the same shape as the input image. Determine first the connected components of this image and then the minimal rectangles containing these components. The hope is that every relevant horizontal/vertical table line is contained in exactly one of the minimal rectangles. We also hope that the minimal rectangles which do not contain a relevant horizontal/vertical table line are exactly those which are too short with respect to the horizontal/vertical direction. (The lower bound for the rectangle length is given by the function argument `rectangle_length_lower_bound`.) The relevant progress image examples for this step are `6`, `7`, `8`, `17`, `18` and `19`.

6) Remove those minimal rectangles constructed in 5) which are too short in the horizontal/vertical direction. If the algorithm works as it is supposed to, there is an exact correspondence between the remaining minimal rectangles and the relevant horizontal/vertical table lines. The relevant progress image examples for this step are `9` and `20`.

7) For each of the remaining minimal rectangles, construct the line segment that is the best fit to the original line-like pixels contained in the rectangle. This line segment will be the ultimate horizontal/vertical table line returned by this function. The variable `horizontal_or_vertical_table_lines` refers to the list containing all of these line segments. The relevant progress image examples for this step are `10`, `11`, `12`, `21`, `22` and `23`.

The code used to construct progress images is rather straightforward and it will not be considered here in detail.

2.5 Function *detect_table_elements*

After the table lines in the input image have been detected, this function is used to detect the table elements in the input image (and to construct images pertaining to this procedure if needed).

A table element is a geometric object which typically represents a local collection of shapes, e.g. a printed or handwritten word, a piece of a larger drawing, or a stain. The construction of a table element is based on the contour detection algorithm provided by `cv2`. (The purpose of this algorithm is to detect shapes or outlines of shapes which can be distinguished clearly from the background.) A more detailed discussion will be given in the following.

Table elements are associated with four types of objects:
1) Printed standard table objects (e.g. column titles)
2) Handwritten entries; there are three subtypes:
  i) Short texts (e.g. numerical entries)
  ii) Long texts (e.g. full text lines and paragraphs)
  iii) Other (e.g. lines crossing out a part of the table)
3) Other semantic objects (e.g. stamps and seals)
4) Unsemantic objects (e.g. stains and tears)

The purpose of this function is to capture all of the relevant information in the input image (except for the table lines), and that is why there is no filtering at this stage. Also, any further analysis of the elements (e.g. the identifying of standard table objects, the combining of the elements associated with a paragraph of text into a higher level semantic object, and so on) will be developed in the future.

Some images illustrating the workings of this function can be found in the directory `./example_images/example_result_images`. We will use the expression `table_elements` to refer to the file `image_3_table_elements.jpg`, and we will do similarly in the case of other files.

The steps of the algorithm are the following:

1) The input image is binarized by using the Otsu method. A great advantage of the Otsu method is that it does not need user-provided parameters.

(The library `cv2` provides a number of binarization methods, and the tests so far showed the Otsu method to be the most suitable.)

2) Remove the table lines determined earlier from the Otsu image by drawing the table lines in the black color. It is essential that the thickness of the removed lines is chosen to be large enough: The table line detection algorithm is not perfect, so increasing the thickness of the removed lines is needed in order to guarantee that the line-like pixels are removed. On the other hand, if the thickness is too large, significant amounts of other pixels will be removed too. In our tests, we have used the value 20 (see `run_main_tests.py`).

3) Detect all contours in the Otsu image. As was mentioned earlier, the contour detection algorithm detects shapes or outlines of shapes which can be distinguished clearly from the background.

4) Draw the detected contours in a zero-initialized image of the same shape as the input image. Once again, we use a relatively high value for the thickness (value 20, see `run_main_tests.py`). The idea is that contours which are close to one another are associated with the same semantic object (e.g. a word or even a paragraph of text), and so such mutually-close contours, when drawn with a large enough thickness, form a blob (more specifically, a connected component) which covers the semantic object mentioned earlier.

5) Determine the connected components in the image constructed in 4). The resulting data structure `table_element_component_parameters` represents the information in the input image which does not pertain to table lines. As was mentioned above, developing methods for analysing the return value of this function is the goal of future work. We will take a closer look at the technical structure of `table_element_component_parameters` when we discuss testing functions in greater detail.

The result image examples `table_elements`, `element_blob_rectangles` and `element_blobs` illustrate the results of this function. The first two of these images are constructed by this function itself and `element_blobs` is constructed by a testing function (see `main_test_functions.py`).

3 Auxiliary functions

The main algorithm uses a number of auxiliary functions and these are divided over a set of files. In this section, we will go over all of the code pertaining to auxiliary functions and not only the parts relevant to the main algorithm.

3.1 File *general_computer_vision_functions.py*

This file contains a number of functions invoking some rather standard and well-known computer vision algorithms provided by `cv2`.

`PARAMETER_DICT`: This is the only global variable in the file. It provides a convenient access to many parameters used by different algorithms in `cv2`. Note that in the current version of the file, `PARAMETER_DICT` is used by only one function. (The function `compute_canny_image` needs the value of four parameters.) During the development of the main algorithm, we experimented with many functions provided by `cv2` and stored some relevant parameter names and values in `PARAMETER_DICT`. We include the full parameter dictionary in order to suggest that this file may be extended by future work.

`triangle_or_otsu_binarization`: Binarization of the input image is often an important step. There are many binarization methods available, but they usually need at least one user-provided parameter. The great advantage of the so-called triangle and Otsu binarization methods is that they do not need any user-provided parameters. The Otsu method seems to work well in the logbook case, which is why it is our chosen binarization method.

`compute_canny_image`: The Canny algorithm is a classical method for detecting edges in an image. During the development of the main algorithm, we did a lot of testing featuring the Canny algorithm. In the current version of the main algorithm, the Canny algorithm is not used, but we still include it in this file, in order give yet another example of a relevant algorithm provided by `cv2`.

`compute_connected_component_parameters`: Detecting connected components in an image is an essential part of the main algorithm. We employ a basic implementation provided by `cv2`. The return value of the function is a list of four objects. The first object in the list is an integer `N` which gives the number of connected components. Since the background is regarded as a component, the number of actual connected components is `N - 1`. The second object in the list is a label array, i.e., an integer array of the same shape as the input image with values `0, 1, 2, ..., N - 1` indicating for each pixel the component containing the pixel. Naturally, the label `0` labels the background pixels. The third object in the list describes for each component the minimal rectangle containing the component. In our discussion on `compute_connected_component_rectangles`, we will take a closer look at this object. The fourth object in the list is a list giving the coordinates of the centroid of each of the components.

`compute_connected_component_rectangles`: Given a connected component computed by `compute_connected_component_parameters`, this function constructs a rectangle containing the component. Initially, the rectangle is the minimal rectangle containing the component, but if needed, the rectangle can be stretched leftwards, rightwards, upwards and/or downwards. (Recall that the rectangles are stretched rightwards/downwards when horizontal/vertical table lines are detected.) Strictly speaking, this function is not a computer vision function but rather a geometric function. However, since the connection between `compute_connected_component_parameters` and this function is very strong, we include the code of this function in the file `general_computer_vision_functions.py`. See the code for the technical details of this very straightforward function.

`detect_contours`: This function detects so-called contours in a binarized input image. We mentioned earlier that the purpose of the contour detection algorithm of `cv2` is to detect shapes or outlines of shapes which can be distinguished clearly from the background. This rather vague characterization of contours is sufficient for our needs. The purpose of the technical arguments `cv.RETR_TREE` and `cv.CHAIN_APPROX_NONE` is to guarantee that the amount of information returned by the function is maximal, and we will not discuss them further here. In addition to the contours themselves, the function returns also the so-called hierarchy of contours. (Basically, contours form a hierarchy based on inclusion relations among the contours. In the current version of the code, contour hierarchies are not used for anything, and so we will not discuss this topic in greater detail.)

`filter_contours`: This function is used to filter the contours detected by `detect_contours`. In fact, this function is not used by the current version of the code, and the purpose of its presence in the code is to remind that contour filtering may be a useful step. This is another example of a function in the file `general_computer_vision_functions.py` which is geometric in nature and not a computer vision function. But due to the strong connection to `detect_contours`, we include the function in the file. In its current form, the function implements a simple filtering based on the size of the area bounded by a given contour.

3.2 File *geometric_operations.py*

This file contains some geometric functions featuring line segments and rectangles.

All of the global variables in this file, namely `FIT_LINE_DISTANCE_PARAMETER`, `FIT_LINE_RADIUS_EPS` and `FIT_LINE_ANGLE_EPS`, are technical arguments used by the `fitLine` function of `cv2` which is invoked once in the function `compute_horizontal_or_vertical_lines_using_rectangles`. The values we have given them are the default values suggested by the `cv2` documentation.

`filter_rectangles`: This is a simple function that filters rectangles based on the lengths of the sides of the rectangles. The code should be rather self-explanatory and will not be discussed in detail here.

`compute_line_point_using_line_element`: The task of this function is to compute a point on a line determined by a given line element. The argument `line_element` is of the form `[delta_x, delta_y, x_0, y_0]` and it determines the line satisfying the equation

$$y - y_0 = \frac{\Delta y}{\Delta x}(x - x_0)$$

where we have used the corresponding mathematical symbols Δx, Δy, x_0 and y_0. The goal is to determine the point `[x_target, y_target]` contained in this line. The idea is that either `x_target` or `y_target` has a numerical value (i.e., is not `None`) as a function argument, and the function is used to compute a numerical value for the other variable. More specifically, if `x_target` is not `None`, the value of `y_target` is given by

$$y_t = \frac{\Delta y}{\Delta x}(x_t - x_0) + y_0,$$

and if `y_target` is not `None`, the value of `x_target` is given by

$$x_t = \frac{\Delta x}{\Delta y}(y_t - y_0) + x_0$$

where x_t and y_t correspond to `x_target` and `y_target`, respectively.

`compute_horizontal_or_vertical_lines_using_rectangles`: This is the main function of the current file. It is invoked by the main algorithm when the actual horizontal/vertical table lines are computed. The setting is the following: We are given a list of rectangles, and the main assumption is that each one of the rectangles contains a table line. The task is to fit a line segment to the line-like pixels in each rectangle. More specifically, the steps are the following for each of the rectangles (see the code itself for more technical details): 1) Determine the coordinates of all of the line-like pixels contained in the rectangle. 2) Use the `fitLine` function of `cv2` to fit a line to the line-like pixels contained in the rectangle. 3) Since the return value of 2) is a variable of the form `line_element`, we need to use `compute_line_point_using_line_element` to compute the endpoints of the line segment representing the relevant table line. In the case of a horizontal table line, we know the values of the x-coordinates of the endpoints and need to compute the y-coordinates. In the case of a vertical table line, the roles of the coordinate axes are switched.

3.3 File *lsd_line_functions.py*

This file contains elementary functions pertaining to `lsd_lines`. Many of these functions are rather redundant from the point of view of the current version of the code, but we include them anyway, since they had their use during initial testing and may have use again in future work.

`study_lsd_lines`: This is a simple (and redundant) function used to study `lsd_lines`. The current version performs a simple study of the `angle` attribute of an `lsd_line` object.

`get_lsd_line_start_point_and_end_point`: This simple function uses attributes of an `lsd_line` to determine its start point and end point.

`study_lsd_lines_and_simple_lines`: This function is used to display pairs of line segments consisting of an `lsd_line` (an object with a number of methods and attributes) and a normal `line` (a pair of two endpoints). The idea is that we have a collection of `lsd_lines` whose elements are transformed in some way (usually a geometric transformation is applied) and the results are represented as normal `lines`. This function takes each pair consisting of the original `lsd_line` and the transformed `line` and displays the pair in an image. Since the function is not essential from the point of view of the main algorithm, we will not study it in detail here. See the discussion on `gui_functions.py` for details as to how functions in `cv2` can be used to display images.

`compute_lsd_line_length`: This simple function computes the length of an `lsd_line`. Note that an `lsd_line` does have the attribute `lineLength`, but this attribute does not necessarily give the right length with respect to the coordinate system of the original input image (we omit the details as to why this can happen).

`filter_lsd_lines`: This function is used by the main algorithm to filter `lsd_lines`. In the context of the main algorithm, an `lsd_line` is filtered if it is too short or not horizontal/vertical enough. (An `lsd_line` is horizontal or vertical enough if `np.abs(np.sin(lsd_line.angle))` or `np.abs(np.cos(lsd_line.angle))` is small enough, respectively.) The full function provides also other filtering conditions. The actual code of the function is rather self-explanatory, so we will not consider it in great detail here.

`lsd_lines_to_horizontal_or_vertical_lines`: The purpose of this function is to transform a collection of `lsd_lines` so that if `lsd_line` is an element of the collection which is nearly horizontal or vertical, then `lsd_line` is mapped to a `line` that goes through the middle point of `lsd_line` and is exactly horizontal or vertical, respectively. We mention that the `pt` attribute of an `lsd_line` gives the coordinates of the middle point of the `lsd_line`, but we will not discuss other details of this functions code, since the function is not used in the current version of the main algorithm.

`draw_lsd_lines`: This simple function can be used to draw `lsd_lines` in an image. It is based on the rather self-explanatory function `line` of `cv2`. The only potentially unintuitive argument is the constant `cv.LINE_AA` – its purpose is to make the drawn lines antialiased.

`draw_lsd_lines_in_octave_images`: This is another function for drawing `lsd_lines`, and it is not used by the current version of the code. It refers to the technical attribute `octave` of `lsd_lines`. From the point of view of the main algorithm, the exact nature of `octaves` is unimportant, so we will not discuss the matter in detail here.

3.4 File *utilities.py*

This file contains some simple auxiliary functions. More precisely, it contains a function for loading images and some functions for drawing geometric shapes.

`load_images`: This function is used to load images in the current version of the code. A whole directory of images is loaded when this function is called. It is essential that the directory contains the image files and nothing else. The most important technical functions assume the input images to be grayscale images, so the `grayscale` argument is typically given the value `True`.

`draw_lines`, `draw_rectangles` and `draw_contours`: These are simple drawing functions which presumably do not have to be discussed in detail. We mention that the meaning of the argument `cv.LINE_AA` is that antialiasing is enabled. Also, the argument `-1` used in `draw_contours` makes the function draw all of the contours in the collection `contours`.

`draw_contour_rectangles`: Instead of drawing a contour itself, this function makes it possible to draw the minimal rectangle containing the contour. Note that this minimal rectangle does not have to be xy-aligned. The function `minAreaRect` constructs the minimal rectangle object. The function `boxPoints` reduces the rectangle object into a collection of corner point coordinates, and the function `intp` turns these coordinates into integers. The meaning of the argument `0` used in `drawContours` is that the function draws the 0-index object in the list `[rectangle]`, i.e., `rectangle` itself.

3.5 File *gui_functions.py*

The functions in this file provide a way to view the images in a list of images one at a time. Typically, the list consists of a base image and versions of the base image which have been obtained after a number of image transformations have been applied.

There are two display modes. In the overview mode, the current image is shown in its totality. In the zoom mode, only a small square-shaped part of the current image is shown, and the user can change the part shown with the keyboard. More specifically, the keyboard commands are the following:

`e`: toggle between the overview mode and the zoom mode
`z`: cycle through the list of images
`w`: move the viewing window up in the zoom mode
`a`: move the viewing window left in the zoom mode
`s`: move the viewing window down in the zoom mode
`d`: move the viewing window right in the zoom mode
`q`: stop viewing the images

`KEY_CODE_DICT`: This dictionary maps letters to keyboard command codes used by `cv2`. There are many more codes included than are used by the current version of the code. This is because we used the functions in this file when calibrating interactively a number of computer vision algorithms. These calibration functions are not included in the current version of the code, but since they or functions like them can be needed in the future, we include the more extensive keyboard command dictionary.

The rest of the global variables pertain to the overview and zoom modes. When in the overview mode, `FULL_VIEW_SCALE_FACTOR` is used to scale down the height and width of the full image being viewed. `VIEW_WINDOW_SIZE` gives the size of the viewing window used while the zoom mode is on. The size of the step taken when the viewing window is moved is given by `VIEW_WINDOW_STEP`. Depending on whether `ZOOM_OUT` is `True` or `False`, the display mode is the overview mode or the zoom mode, respectively. The variables `X_MIN` and `Y_MIN` give the current coordinates of the top-left corner point of the viewing window.

`display_image`: This is the secondary function in this file, and it is called by the primary function `display_multiple_images`. This function is used to toggle between the overview mode and the zoom mode, to move the viewing window around while the zoom mode is on, and to determine and display the current image. The variable `image_to_display` refers to the current image being displayed. It is either a scaled-down version of the full image or the part of the full image determined by the viewing window. See the code itself for further technical details.

`display_multiple_images`: This is the primary function of the file, and it is the one called from outside the file even if the number of images to show is one. (See the next paragraph for a technical comment about using `cv2` to display images.) In addition to calling `display_image` in order to display the current image, this function detects keyboard commands (this is done using `waitKey`) and makes it possible for the user to cycle through the relevant list of images or to stop viewing the images altogether.

The minimal code needed to display an `image` described by `title` consists essentially of the lines `cv.imshow(title, image)` and `cv.waitKey(0)`. The command `cv.waitKey(0)` makes the program wait for a keyboard command for an indefinite amount of time. It can be replaced by other commands, see the `cv2` documentation for details on this, but the essential point is that it is not sufficient to call `imshow` alone. To display an image, some GUI housekeeping tasks need to be done, and this is not accomplished by calling `imshow`.

3.6 File *numpy_array_operations.py*

The current version of the main algorithm produces as its results some images and `numpy` arrays. This file contains some technical functions which are used in the context of these `numpy` arrays.

`construct_compressed_array`: This function compresses a `numpy` array to some extent by extracting the non-zero values and the corresponding row and column indices. The result is a `numpy` array with three columns such that the first row contains the height and width of the input array, and each of the other rows contains the index and value information pertaining to one non-zero value of the input array. See the code for the technical details. (At the moment, `numpy` does not have sparse matrices implemented, and we did not want to introduce a new library, for example `scipy` or `tensorflow`, in order to have them, and so we wrote our own rather simple function.)

`construct_array_from_compressed_array`: This function is simply the inverse function of `construct_compressed_array`.

`construct_label_image`: Recall that the general computer vision function `compute_connected_component_parameters` returns a list of four objects such that the second object is a label array containing information about the connected components of the input image. More specifically, the label array has as values the labels `0, 1, 2, ..., N - 1`, where `0` corresponds to the background and the other labels to the connected components of the input image, so that the label array gives for each pixel the connected component containing the pixel. The purpose of the discussed function is to construct an image corresponding to the label array, see `image_5_element_blobs.jpg` in `./example_images/example_result_images`. The current code of the function is simple and probably relatively fast, but the downside is that any component whose label is a multiple of 256 will be assigned the background color (i.e., black), because the construction is based on the `mod 256` operation. However, since the downside rarely affects more than two components in a given input image, we regard it as a problem that can be ignored for the time being.

`construct_label_image_slow_version`: This is a slower version of the function `construct_label_image` which does not have the downside mentioned above. Since this version is not used by the current version of the code, we will not discuss it further here.

4 Analysis functions

The current versions of the table line and element detection functions are preliminary, so developing sophisticated analysis functions does not make sense at this stage. However, we do want to give an example of a relatively simple analysis function which nonetheless yields rather nice basic results.

The purpose of the analysis function is to place a given table element into the appropriate table cell. Instead of working with the actual table lines obtained from the main algorithm, we make two simplifications. First of all, we replace each horizontal/vertical table line – which almost always is not exactly horizontal/vertical – with an exactly horizontal/vertical line segment that goes through the middle point of the table line. Additionally, we extend these line segments to be maximal. So instead of considering the actual table, we consider an exact xy-grid that approximates the actual table, and place the table elements into the cells of this xy-grid. The placing of a given table element is done in the following way: The given table element is contained in a minimal rectangle. Place the element into the grid cell which contains the center point of this rectangle.

See `./example_images/example_analysis_images` for some examples. In the example images, the minimal rectangles of table elements which have been placed in the same grid cell have the same color. We can see that even this rather simple method works well in the case of small table elements.

Let us take a closer look at the code itself.

4.1 File *analysis_functions.py*

The main function of this file is `table_element_position_analysis`, but we will first go through the auxiliary functions. (The global variables in this file are of the kind we have encountered before, i.e., they are used in the construction of images which illustrate the results of the main function, and so we will not consider them in detail.)

`compute_sorted_mean_coordinates`: This function computes a sorted collection of mean coordinates for a given set of horizontal/vertical table lines. If the lines are horizontal, the coordinates are y-coordinates, and the coordinates are x-coordinates in case the lines are vertical. More specifically, for a horizontal table line, the computed coordinate is the y-coordinate of the middle point of the table line, and for a vertical table line, we get the x-coordinate of the middle point. Basically, this function is used to compute the positions of the lines in the xy-grid that approximates the actual table. The computed coordinates are sorted, since this is important for future steps of the algorithm. Additionally, if 0 is not in the resulting collection of coordinates, it is added into the collection. This is done in order to guarantee that a given table element will always have a grid line on its left side and another one above it.

`determine_cell_line`: Assuming that a given table element is in a given grid cell, one can use this function to determine the grid lines of the cell (called cell lines). More specifically, the left cell line and the top cell line are explicitly determined using this function. Let us consider the case of the left cell line only, since the case of the top cell line is similar. In this case, the given collection of sorted mean coordinates contains the x-positions of the vertical grid lines. The `coordinate` argument is the x-coordinate of the center point `p` of the minimal rectangle containing the given table element. We first compute the signed distances of `p` from the vertical grid lines. Let `d_min` be the signed distance with the minimal absolute value, and let `L` be the grid line closest to `p` (so the signed distance between `p` and `L` is `d_min`). If `d_min` is negative, it means that `L` is on the left-hand side of `p` and, therefore, `L` is the left cell line we are looking for. If `d_min` is positive, then `L` is on the right-hand side of `p`, and so the left cell line we want is the grid line on the left-hand side of `L`.

`determine_table_element_cell_positions`: This function determines the appropriate grid cell for each table element. In technical terms, the following takes place. Let `p` be the center point of the minimal rectangle containing the given table element. We use `determine_cell_line` to determine the top and left cell lines of `p`. The top and left cell lines determine a unique cell, and so we can place `p` in this cell and, therefore, also the element itself and its minimal rectangle. In the code, we associate the minimal rectangle with the cell because this facilitates the construction of an image that describes the results of the algorithm.

`construct_table_element_cell_position_image`: This simple function is used to construct images which illustrate the results of the main function.

`table_element_position_analysis`: This is the main function in the file. Given our discussion on both the general idea behind the analysis method and the auxiliary functions, the code should be easy to understand. In order to emphasize the preliminary nature of this function, the function returns only an image, i.e., more abstract objects describing the results are not returned.

5 Testing functions

The current version of the code is preliminary in nature, and therefore the code is under development. This is why the testing has been done using separate scripts while the algorithm scripts (and even the testing scripts themselves) were being developed.

The code relevant to testing will be discussed in the subsections below. Here we will take a look at the directory structures pertaining to testing data and testing results.

The testing data root directory is `./sample_logbook_data`. This directory contains a subdirectory for each sample logbook and nothing else. The name of the subdirectory associated with a logbook will be used as the name of the logbook.

We were given 125 sample logbooks altogether and we ended up naming them simply by using the integers `1, 2, 3, ..., 125`. We have included a handful of them in the current version of `./sample_logbook_data`.

The directory associated with a logbook contains a number of subdirectories, each of which contains pages of different types, e.g. logbook table pages with handwritten markings, empty logbook table pages, or pages which are printed but not table pages. The idea is that every image file associated with the logbook is included in exactly one of these subdirectries. The naming scheme for a logbook page is `{logbook}_{page_number}.jpg`. (The page numbers used here are integers obtained from an enumeration of the image files.)

The only subdirectory associated with a logbook that is relevant for the current version of the code is `logbook_pages`. It contains the logbook table pages that have handwritten markings. We have included some sample pages for each of the sample logbooks.

The root directory for testing results is `./sample_logbook_results`. For a logbook named `logbook` processed by the main algorithm, a subdirectory `logbook` will created. This subdirectory will contain a number of subsubdirectories for storing results of different kinds. The subsubdirectories are `arrays`, `images` and `progress_images`. The files that will be stored in them will be discussed in detail later.

5.1 File *run_main_tests.py*

This file contains the code for launching the main test functions. There are two main test functions. There is `random_sample_test` which chooses a random page of a random logbook, processes it, and displays the results as images onscreen. The other main test function is `multiple_logbooks_test` which processes all of the logbooks contained in `./sample_logbook_data` (or more generally, in the root data directory) and saves the results in `./sample_logbook_results` (or more generally, in the root result directory).

The file contains many global variables. Some of them pertain to the main algorithm. Some of them determine which result images will be constructed and saved (in the current version of the code, all result arrays are always saved). Also, there are many which are concerned with data and result directories and filenames. The nature of each of these variables should be either self-explanatory or otherwise clear given our presentation hitherto. For example, if a variable gives a value to some function argument, the name of the variable is exactly the same as the name of the function argument.

After the global variables have been introduced, they are collected into lists which will serve as collective function arguments which will be unpacked when appropriate. Finally, the selected main test function is launched.

5.2 File *main_test_functions.py*

The functions `random_sample_test` and `multiple_logbooks_test` are the main test functions in this file. Additionally, there are many auxiliary functions pertaining to, for example, loading images, creating save directories, preparing and saving result arrays and images, and printing messages relevant to the test in progress.

Regarding saving result arrays and images, we mention that we do not employ any official compression methods, since we want to get an idea as to the sizes of the files. (We do employ a compression method of our own in one instance. This is the function `construct_compressed_array` contained in numpy_array_operations.py.)

construct_logbook_list: This simple function constructs a list containing all of the logbooks to be processed by multiple_logbooks_test. It is essentially trivial, but the reasoning for its existence follows from our decision not to do any collective function argument unpacking in the main test functions themselves.

load_logbook_page_images: This function loads the input images associated with a given logbook.

load_random_logbook_page_image: The function random_sample_test uses this function to load the images it processes one at a time.

create_save_directories: For a given logbook, this function creates the required save directories.

save_result_arrays: This is a straightforward function for saving result arrays.

save_result_images: This is a function for saving result images. The list result_images has already been prepared by prepare_result_images (see the description of this function later in this subsection), and so only the corresponding list of file suffixes has to be constructed, which is a simple task. The variable result_image_number is used in filename in order to guarantee that the resulting result image files have a certain appropriate order.

save_progress_images: This function saves progress images related to the detection of table lines, see ./example_images/example_progress_images. The reason we make this function return a time variable pertaining to the function operation is that we want to avoid unpacking collective function arguments in the main test functions.

study_and_save_numbers_of_table_elements: A result array is almost always associated with a particular logbook page. The exceptional result array is numbers_of_table_elements which is associated with a given logbook in its totality. In terms of structure, the array consists of rows which are of the form [image_number, number_of_table_elements], so this array simply records how many table elements were detected in each page. In addition to saving the array, the function being discussed also computes the maximum and the mean over the numbers of table elements contained in the array and prints out the result.

`prepare_result_arrays`: This function prepares the page-specific result arrays for saving and extends `numbers_of_table_elements` appropriately (see the the discussion on `study_and_save_numbers_of_table_elements` for details). There are six page-specific result arrays. Three of them pertain to table lines: one contains the horizontal, another the vertical, and the third all of the table lines. (Recall that a table line is represented by a pair of endpoints expressed in the coordinate system of the input image.) The other three pertain to the table elements of the input image. (Recall that the current algorithm identifies table elements with certain connected components.) There is the label array (which we save in a compressed form) that gives for each pixel in the input image the table element containing the pixel. There is also an array describing the minimal rectangles containing the table elements. Such a rectangle is represented by four numbers: the x and y-coordinates of the top-left corner point, and the x and y-lengths of the rectangle (i.e., the width and the height). Finally, there is an array which stores for each table element the coordinates (in the coordinate system of the input image) of the centroid of the element. Note that the rectangle and centroid arrays are saved in their unprocessed forms, i.e., the rectangle description format is not the same as used by, e.g., drawing functions, and both rectangles contain on their first rows information about the background of the input image. We do this because, in the current version of the code, rectangles are processed locally when needed, and the centroids are not used at all.

`prepare_result_images`: This function prepares result images to be saved, see `./example_images/example_result_images` for a set of examples. A full set of result images contains six images. The first one is the original input image (in grayscale format). The second image displays the table lines found by the main algorithm. The next three are constructed by the table element detection algorithm. The first one of these shows the input image with the minimal table element rectangles drawn in. Next we have a binary image showing the blobs (i.e., connected components) corresponding to the table elements, and also the minimal rectangles. The last image in this set of three displays the blobs using different colors. This image is constructed by the discussed function by letting `construct_label_image` of `numpy_array_operations.py` act on `element_label_array` (an array considered in detail in the previous paragraph). The final result image shows again the input image with the minimal element rectangles added. This time the colors of the rectangles vary. The idea is that elements associated with the same table cell share rectangle color. This image is created by `table_element_position_analysis` contained in `analysis_functions.py`. See the previous section for more details.

`determine_images_to_display`: This function determines the images to display after a `random_sample_test` has been performed. This function is very similar to `prepare_result_images`, so we will not consider it in greater detail.

`print_random_sample_test_times`: This simple, self-explanatory function prints information pertaining to a run of `random_sample_test`.

`print_multiple_logbooks_test_times`: This function prints information after each time `multiple_logbooks_test` has processed an input image. The construction of the string printed is more complex than in the function `print_random_sample_test_times` but still rather simple.

`print_logbook_total_time`: This simple function prints the elapsed time after `multiple_logbooks_test` has processed a whole logbook.

`random_sample_test`: This is the first main test function. It processes random pages of random logbooks one at a time and displays the results onscreen. Given our discussion on the auxiliary functions above, the code of the function should be easy to understand.

`multiple_logbooks_test`: This is the second main test function. It processes all of the logbooks in `./sample_logbook_data`, or more generally, in the root data directory, and saves the results in `sample_logbook_results`, or more generally, in the root result directory. Given our discussion on the auxiliary functions above, the code of the function should be easy to understand.