Chapter 6

Image/Document Alignment and Registration

In this chapter, you will learn how to perform image alignment and image registration using OpenCV. Image alignment and registration have several practical, real-world use cases, including:

- **Medical:** Magnetic resonance imaging (MRI) scans, single-photon emission computerized tomography (SPECT) scans, and other medical scans produce multiple images. Image registration can *align* multiple images together and overlay them on top of each other to help doctors and physicians better interpret these scans. From there, the doctor can read the results and provide a more accurate diagnosis.
- **Military:** Automatic target recognition (ATR) algorithms accept multiple input images of the target, align them, and refine their internal parameters to improve target recognition.
- **Optical character recognition (OCR):** Image alignment (often called document alignment in the context of OCR) can be used to build automatic form, invoice, or receipt scanners. We first align the input image to a template of the document we want to scan. From there, OCR algorithms can read the text from each field.

In this chapter's context, we'll be looking at image alignment through the perspective of **document alignment/registration**, which is often used in OCR applications.

This chapter will cover the fundamentals of image registration and alignment. Then, in the next chapter, we'll incorporate image alignment with OCR, allowing us to create a document, form, and invoice scanner that aligns an input image with a template document and then extracts the text from each field in the document.

6.1 Chapter Learning Objectives

In this chapter, you will:

- Learn how to detect keypoints and extract local invariant descriptors using oriented FAST and rotated BRIEF (ORB)
- Discover how to match keypoints together and draw correspondences
- Compute a homography matrix using the correspondences
- Take the homography matrix and use it to apply a perspective warp, thereby aligning the two images/documents together

6.2 Document Alignment and Registration with OpenCV

In the first part of this chapter, we'll briefly discuss what image alignment and registration is. We'll learn how OpenCV can help us align and register our images using keypoint detectors, local invariant descriptors, and keypoint matching.

Next, we'll implement a helper function, `align_images`, which, as the name suggests, will allow us to align two images based on keypoint correspondences.

I'll then show you how to use the `align_document` function to align an input image with a template.

6.2.1 What Is Document Alignment and Registration?

Image alignment and registration is the process of:

i. Accepting two input images that contain the *same* object but at slightly different *viewing angles*

ii. Automatically computing the homography matrix used to align the images (whether that be featured-based keypoint correspondences, similarity measures, or even deep neural networks that *automatically* learn the transformation)

iii. Taking that homography matrix and applying a perspective warp to align the images together

For example, consider Figure 6.1. On the *top-left*, we have a template of a W-4 form, which is a U.S. Internal Revenue Service (IRS) tax form that employees fill out so that employers know how much tax to withhold from their paycheck (depending on deductions, filing status, etc.).

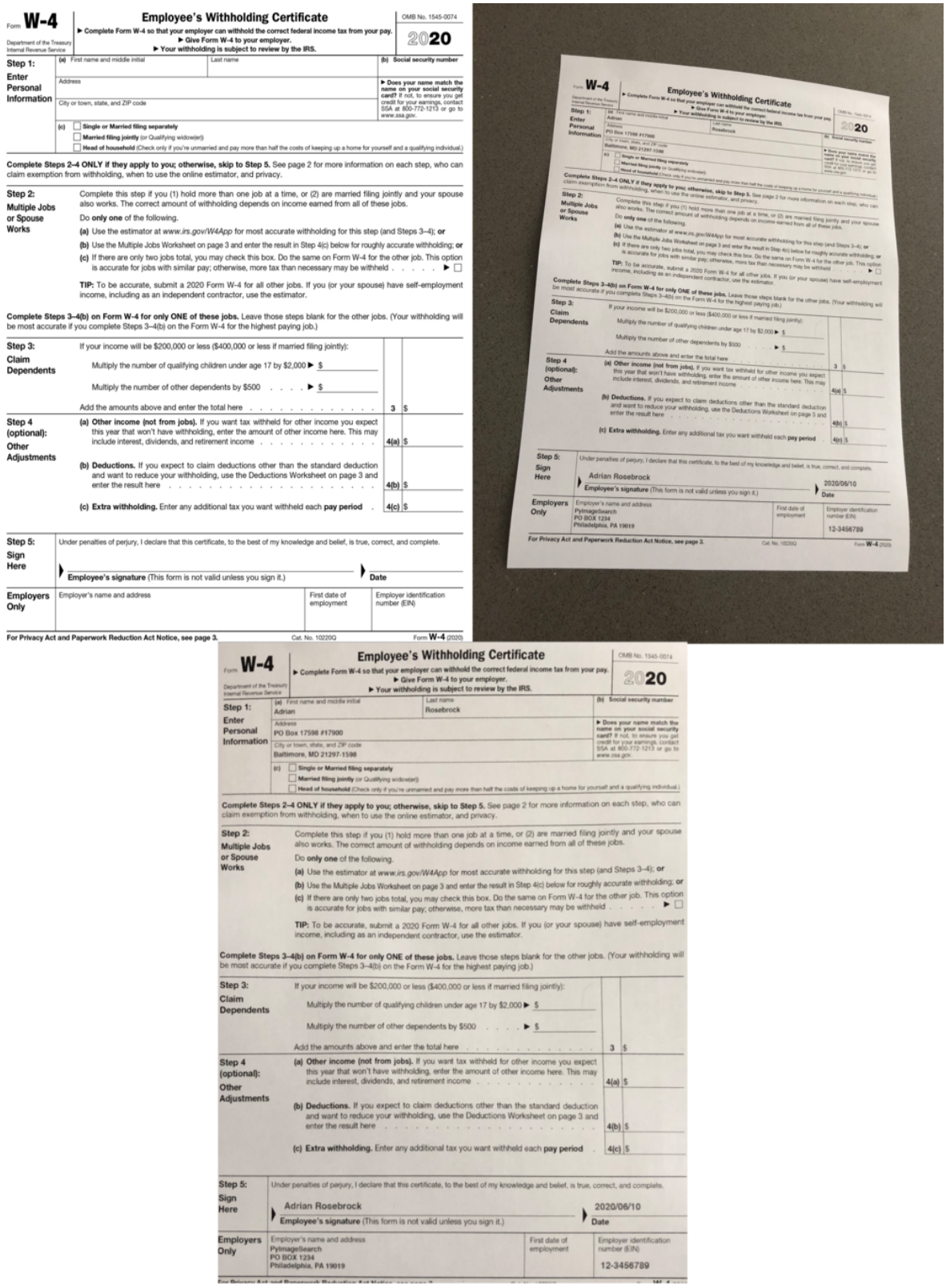


Figure 6.1. We have three scans of the well-known IRS W-4 form. We have an empty form *(top-left)*, a partially completed form that is out of alignment *(top-right)*, and our processed form that has been aligned by our algorithm *(bottom)*.

I have partially filled out a W-4 form with faux data and then captured a photo of it with my phone *(top-right)*.

Finally, you can see the output image alignment and registration on the bottom — **notice how the input image is now aligned with the template.**

In the next chapter, you'll learn how to OCR each of the individual fields from the input document and associate them with the template fields. For now, though, we'll only be learning how to align a form with its template as an important pre-processing step before applying OCR.

6.2.2 How Can OpenCV Be Used for Document Alignment?

There are several image alignment and registration algorithms:

- The most popular image alignment algorithms are **feature-based**. They include keypoint detectors (DoG, Harris, good features to track, etc.), local invariant descriptors (SIFT, SURF, ORB, etc.), and keypoint matching such as random sample consensus (RANSAC) and its variants.

- Medical applications often use **similarity measures** for image registration, typically cross-correlation, the sum of squared intensity differences, and mutual information.

- With the resurgence of neural networks, **deep learning** can even be used for image alignment by automatically learning the homography transform.

We'll be implementing image alignment and registration using feature-based methods.

Feature-based methods start with detecting keypoints in our two input images (Figure 6.2).

Keypoints are meant to identify salient regions of an input image. We extract local invariant descriptors for each keypoint, quantifying the region surrounding each keypoint in the input image.

Scale-invariant feature transform (SIFT) features, for example, are 128-d, so if we detect 528 keypoints in a given input image, then we'll have a total of 528 vectors, each of which is 128-d.

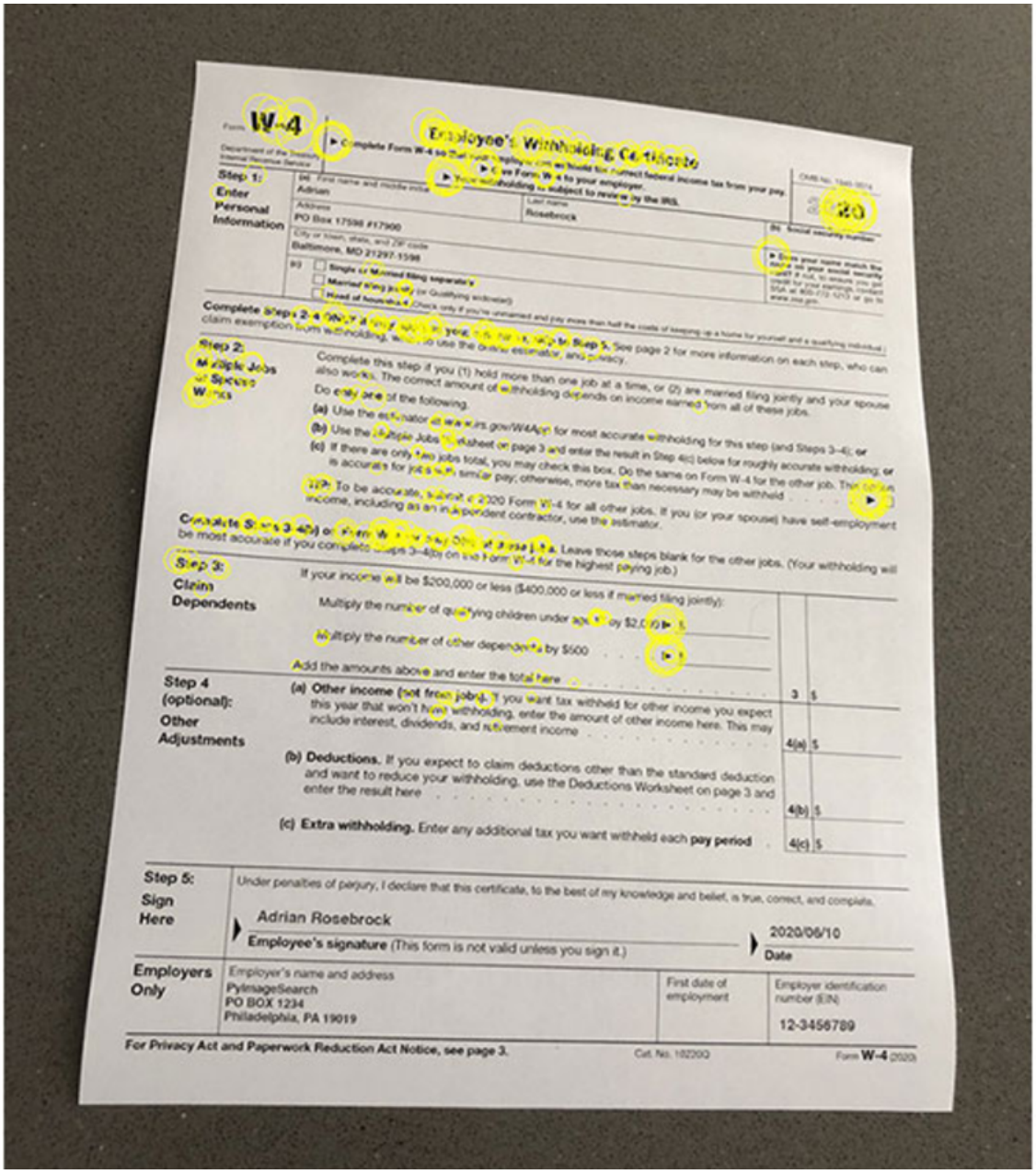


Figure 6.2. We use a feature-based image alignment technique that detects keypoints that are the location of regions of interest. The keypoints found are highlighted in yellow. They will be used to align and register the two images (documents/forms).

Given our features, we apply algorithms such as RANSAC [18] to match our keypoints and determine their correspondences (Figure 6.3).

Provided we have enough keypoint matches and correspondences. We can then compute a homography matrix, which allows us to apply a perspective warp to align the images. You'll be learning how to build an OpenCV project that accomplishes image alignment and registration via a homography matrix in the remainder of this chapter.

For more details on homography matrix construction and the role it plays in computer vision, be sure to refer to this OpenCV reference: http://pyimg.co/ogkky [19].

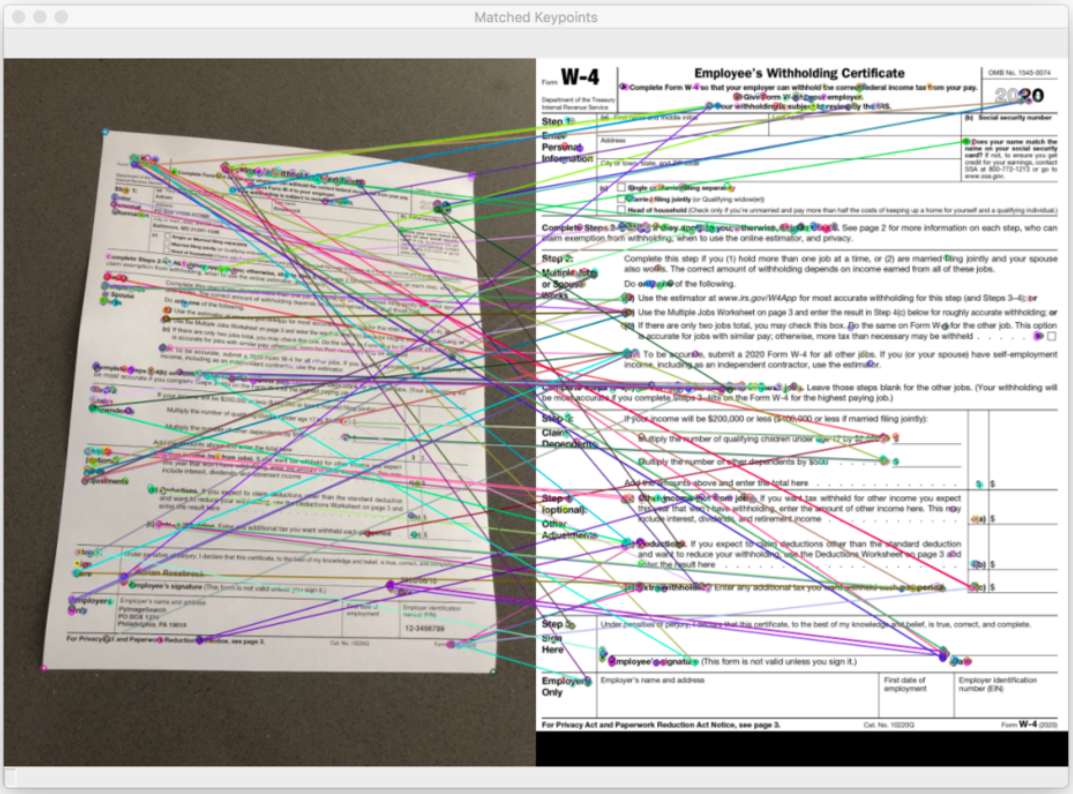


Figure 6.3. Here we see the keypoints matched between our partially completed form that is out of alignment with our original form that is properly aligned. You can see there are colored lines drawn using OpenCV between each corresponding keypoint pair.

6.2.3 Project Structure

Let's start by inspecting our project directory structure:

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- alignment
|   |   |-- __init__.py
|   |   |-- align_images.py
|-- forms
|   |-- form_w4.pdf
|   |-- form_w4_orig.pdf
|-- scans
|   |-- scan_01.jpg
|   |-- scan_02.jpg
```

```
|-- align_document.py
|-- form_w4.png
```

We have a simple project structure consisting of the following images:

- `scans/`: Contains two JPG testing photos of a tax form
- `form_w4.png`: Our template image of the official 2020 IRS W-4 form

Additionally, we'll be reviewing two Python files:

- `align_images.py`: Holds our helper function, which aligns a scan to a template utilizing an OpenCV pipeline
- `align_document.py`: Our driver file in the main directory brings all the pieces together to perform image alignment and registration with OpenCV

In the next section, we'll work on implementing our helper utility for aligning images.

6.2.4 Aligning Documents with Keypoint Matching

We are now ready to implement image alignment and registration using OpenCV. For this section, we'll attempt to align the images in Figure 6.4.

On the *left*, we have our template W-4 form, while on the *right*, we have a sample W-4 form I have filled out and captured with my phone. The end goal is to align these images such that their fields match up.

Let's get started! Open `align_images.py` inside the `alignment` submodule of `pyimagesearch` and let's get to work:

```
1  # import the necessary packages
2  import numpy as np
3  import imutils
4  import cv2
5
6  def align_images(image, template, maxFeatures=500, keepPercent=0.2,
7      debug=False):
8      # convert both the input image and template to grayscale
9      imageGray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
10     templateGray = cv2.cvtColor(template, cv2.COLOR_BGR2GRAY)
```

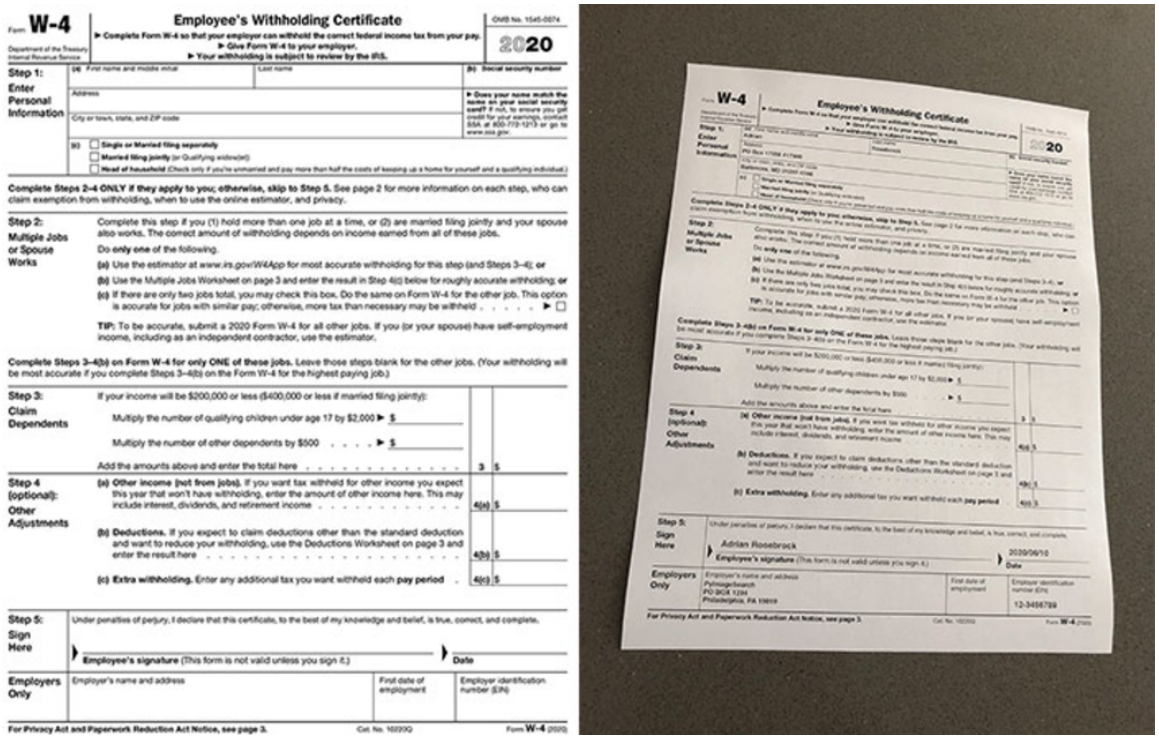


Figure 6.4. We have two similar W-4 forms in different orientations. We have our original template form in the desired orientation *(left)* and the partially completed form that needs to be aligned *(right)*. Our goal is to use OpenCV to align the *right* image to the *left* template image using keypoint matching and a homography matrix.

Our `align_images` function begins on **Line 6** and accepts five parameters:

- `image`: Our input photo/scan of a form (e.g., IRS W-4). From an arbitrary viewpoint, the form itself should be identical to the `template` image but with form data present.
- `template`: The template form image.
- `maxFeatures`: Places an upper bound on the number of candidate keypoint regions to consider.
- `keepPercent`: Designates the percentage of keypoint matches to keep, effectively allowing us to eliminate noisy keypoint matching results
- `debug`: A flag indicating whether to display the matched keypoints. By default, keypoints are not displayed. However, I recommend setting this value to `True` for debugging purposes.

Given that we have defined our function let's implement our image processing pipeline. Diving in, the first action we take is converting both our `image` and `template` to grayscale (**Lines 9 and 10**).

Next, we will detect keypoints, extract local binary features, and correlate these features between our input image and the template:

```
12      # use ORB to detect keypoints and extract (binary) local
13      # invariant features
14      orb = cv2.ORB_create(maxFeatures)
15      (kpsA, descsA) = orb.detectAndCompute(imageGray, None)
16      (kpsB, descsB) = orb.detectAndCompute(templateGray, None)
17
18      # match the features
19      method = cv2.DESCRIPTOR_MATCHER_BRUTEFORCE_HAMMING
20      matcher = cv2.DescriptorMatcher_create(method)
21      matches = matcher.match(descsA, descsB, None)
```

We use the ORB algorithm (http://pyimg.co/46f2s [20]) to detect keypoints and extract binary local invariant features (**Lines 14–16**). The Hamming method computes the distance between these binary features to find the best matches (**Lines 19–21**). You can learn more about the keypoint detection and local binary patterns in my *Local Binary Patterns with Python & OpenCV* tutorial (http://pyimg.co/94p11 [21]) or the PyImageSearch Gurus course (http://pyimg.co/gurus [22]).

As we now have our keypoint `matches`, our next steps include sorting, filtering, and displaying:

```
23      # sort the matches by their distance (the smaller the distance,
24      # the "more similar" the features are)
25      matches = sorted(matches, key=lambda x:x.distance)
26
27      # keep only the top matches
28      keep = int(len(matches) * keepPercent)
29      matches = matches[:keep]
30
31      # check to see if we should visualize the matched keypoints
32      if debug:
33          matchedVis = cv2.drawMatches(image, kpsA, template, kpsB,
34              matches, None)
35          matchedVis = imutils.resize(matchedVis, width=1000)
36          cv2.imshow("Matched Keypoints", matchedVis)
37          cv2.waitKey(0)
```

Here, we sort the `matches` (**Line 25**) by their distance. The *smaller* the distance, the *more similar* the two keypoint regions are. **Lines 28 and 29** keep only the top matches — otherwise we risk introducing noise.

If we are in `debug` mode, we will use `cv2.drawMatches` to visualize the matches using OpenCV drawing methods (**Lines 32–37**).

Next, we will conduct a couple of steps before computing our homography matrix:

```
39      # allocate memory for the keypoints (x,y-coordinates) from the
40      # top matches -- we'll use these coordinates to compute our
41      # homography matrix
42      ptsA = np.zeros((len(matches), 2), dtype="float")
43      ptsB = np.zeros((len(matches), 2), dtype="float")
44
45      # loop over the top matches
46      for (i, m) in enumerate(matches):
47          # indicate that the two keypoints in the respective images
48          # map to each other
49          ptsA[i] = kpsA[m.queryIdx].pt
50          ptsB[i] = kpsB[m.trainIdx].pt
```

Here we are:

- Allocating memory to store the keypoints (`ptsA` and `ptsB`) for our top matches
- Initiating a loop over our top `matches` and inside indicating that keypoints A and B map to one another

Given our organized pairs of keypoint matches, now we're ready to align our image:

```
52      # compute the homography matrix between the two sets of matched
53      # points
54      (H, mask) = cv2.findHomography(ptsA, ptsB, method=cv2.RANSAC)
55
56      # use the homography matrix to align the images
57      (h, w) = template.shape[:2]
58      aligned = cv2.warpPerspective(image, H, (w, h))
59
60      # return the aligned image
61      return aligned
```

Aligning our image can be boiled down to our final two steps:

- Find our homography matrix using the keypoints and RANSAC algorithm (**Line 54**).
- Align our image by applying a warped perspective (`cv2.warpPerspective`) to our `image` and matrix, `H` (**Lines 57 and 58**). This `aligned` image result is returned to the caller via **Line 61**.

Congratulations! You have completed the most technical part of the chapter.

Note: *A big thanks to Satya Mallick at LearnOpenCV for his concise implementation of keypoint matching (http://pyimg.co/16qi2 [23]), upon which ours is based.*

6.2.5 Implementing Our Document Alignment Script

Now that we have the `align_images` function at our disposal, we need to develop a driver script that:

i. Loads an image and template from disk

ii. Performs image alignment and registration

iii. Displays the aligned images to our screen to verify that our image registration process is working properly

Open `align_document.py`, and let's review it to see how we can accomplish exactly that:

```
1  # import the necessary packages
2  from pyimagesearch.alignment import align_images
3  import numpy as np
4  import argparse
5  import imutils
6  import cv2
7
8  # construct the argument parser and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--image", required=True,
11     help="path to input image that we'll align to template")
12 ap.add_argument("-t", "--template", required=True,
13     help="path to input template image")
14 args = vars(ap.parse_args())
```

The key import on **Lines 2–6** that should stand out is `align_images`, a function which we implemented in the previous section.

Our script requires two command line arguments:

- `--image`: The path to the input image scan or photo
- `--template`: Our template image path; this could be an official company form or in our case the 2020 IRS W-4 template image

Our next step is to align our two input images:

```
16 # load the input image and template from disk
17 print("[INFO] loading images...")
18 image = cv2.imread(args["image"])
19 template = cv2.imread(args["template"])
20
```

```
21 # align the images
22 print("[INFO] aligning images...")
23 aligned = align_images(image, template, debug=True)
```

After loading both our input `--image` and input `--template` (**Lines 18 and 19**), we take advantage of our helper routine, `align_images`, passing each as a parameter (**Line 23**).

Notice how I've set the `debug` flag to `True`, indicating that I'd like the matches to be annotated. When you make the `align_images` function part of a real OCR pipeline you will turn the debugging option off.

We're going to visualize our results in two ways:

i. **Stacked** side-by-side

ii. **Overlayed** on top of one another

These visual representations of our results will allow us to determine if the alignment is/was successful.

Let's prepare our `aligned` image for a *stacked* comparison with its template:

```
25 # resize both the aligned and template images so we can easily
26 # visualize them on our screen
27 aligned = imutils.resize(aligned, width=700)
28 template = imutils.resize(template, width=700)
29
30 # our first output visualization of the image alignment will be a
31 # side-by-side comparison of the output aligned image and the
32 # template
33 stacked = np.hstack([aligned, template])
```

Lines 27 and 28 resize the two images such that they will fit on our screen. We then use `np.hstack` to *stack our images next to each other* to easily inspect the results (**Line 33**).

And now let's *overlay* the `template` form on top of the `aligned` image:

```
35 # our second image alignment visualization will be *overlaying* the
36 # aligned image on the template, that way we can obtain an idea of
37 # how good our image alignment is
38 overlay = template.copy()
39 output = aligned.copy()
40 cv2.addWeighted(overlay, 0.5, output, 0.5, 0, output)
41
42 # show the two output image alignment visualizations
43 cv2.imshow("Image Alignment Stacked", stacked)
```

```
44 cv2.imshow("Image Alignment Overlay", output)
45 cv2.waitKey(0)
```

In addition to the side-by-side stacked visualization from above, an alternative visualization is to *overlay* the input image on the template, so we can readily see the amount of misalignment. **Lines 38–40** use OpenCV's `cv2.addWeighted` to transparently blend the two images into a single `output` image with the pixels from each image having equal weight.

Finally, we display our two visualizations on-screen (**Lines 43–45**).

Well done! It is now time to inspect our results.

6.2.6 Image and Document Alignment Results

We are now ready to apply image alignment and registration using OpenCV! Open a terminal and execute the following command:

```
$ python align_document.py --template form_w4.png --image scans/scan_01.jpg
[INFO] loading images...
[INFO] aligning images...
```

Figure 6.5 *(top)* shows our input image, `scan_01.jpg` — notice how this image has been captured with my smartphone at a non-90° viewing angle (i.e., not a *top-down*, bird's-eye view of the input image).

It's very common for images to be captured from unpredictable angles when capturing documents with phone cameras.

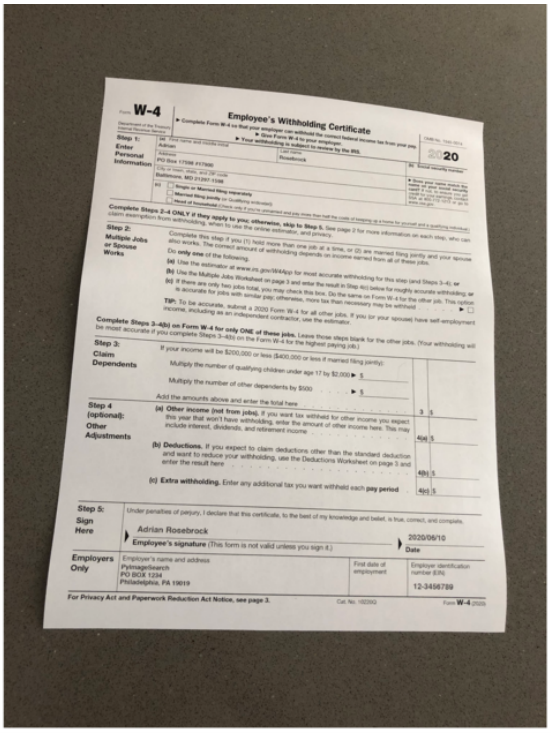
To get better light, people will often move to the side of an image creating an angle to their photo.

Many times people won't even be aware they are taking a picture at an angle. This means many images are captured from an angle. For OCR to work, we need a way to allow people to take photos from multiple angles. We are the ones who need to align the images after they are taken.

To create an OCR pipeline that doesn't do this alignment would result in a brittle and inaccurate system.

We then apply image alignment and registration, resulting in Figure 6.5 *(bottom)*.

On the *bottom-left*, you can see the input image (after alignment), while the *bottom-right* shows the original W-4 template image. Notice how the two images have been automatically aligned using keypoint matching!



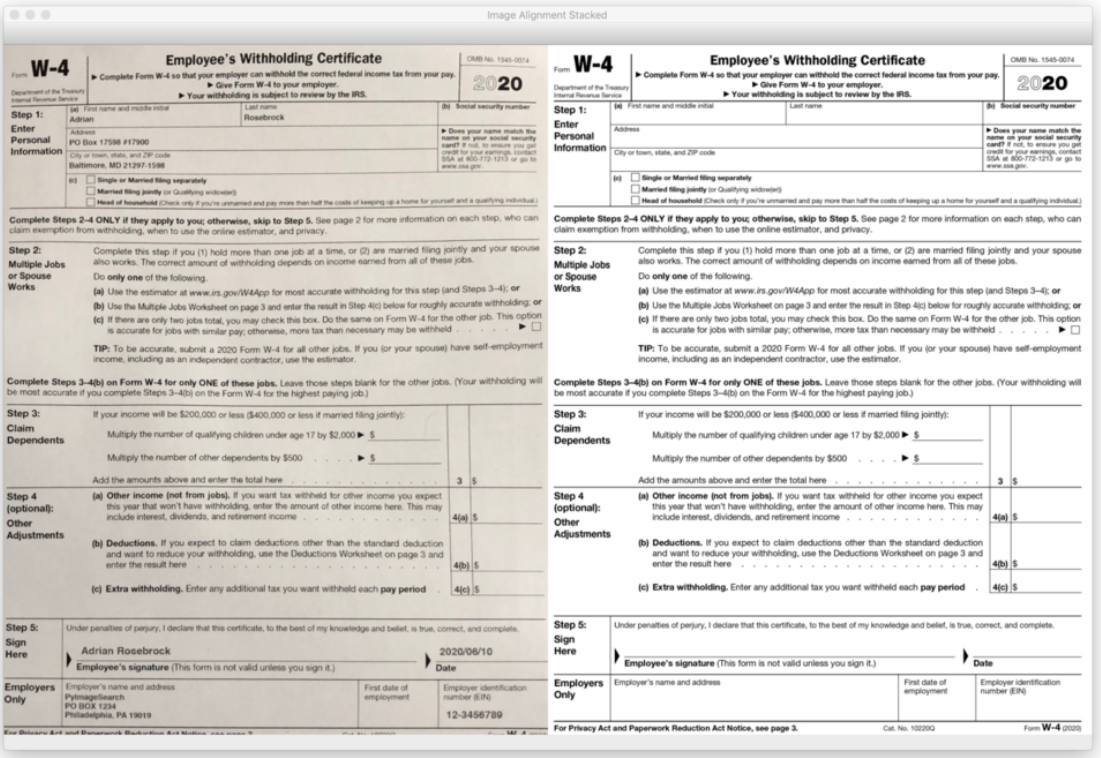


Figure 6.5. *Top:* We have our partially completed form, which has been taken with our iPhone from the perspective of looking down at the image on the table. The resulting image is at an angle and rotated. *Bottom:* We have a side-by-side comparison. Our partially completed form has been aligned and registered *(left)* to be more similar to our template form *(right)*. Our OpenCV keypoint matching algorithm did a pretty nice job!

An alternative visualization can be seen in Figure 6.6. Here we have overlayed the output aligned image on top of the template.

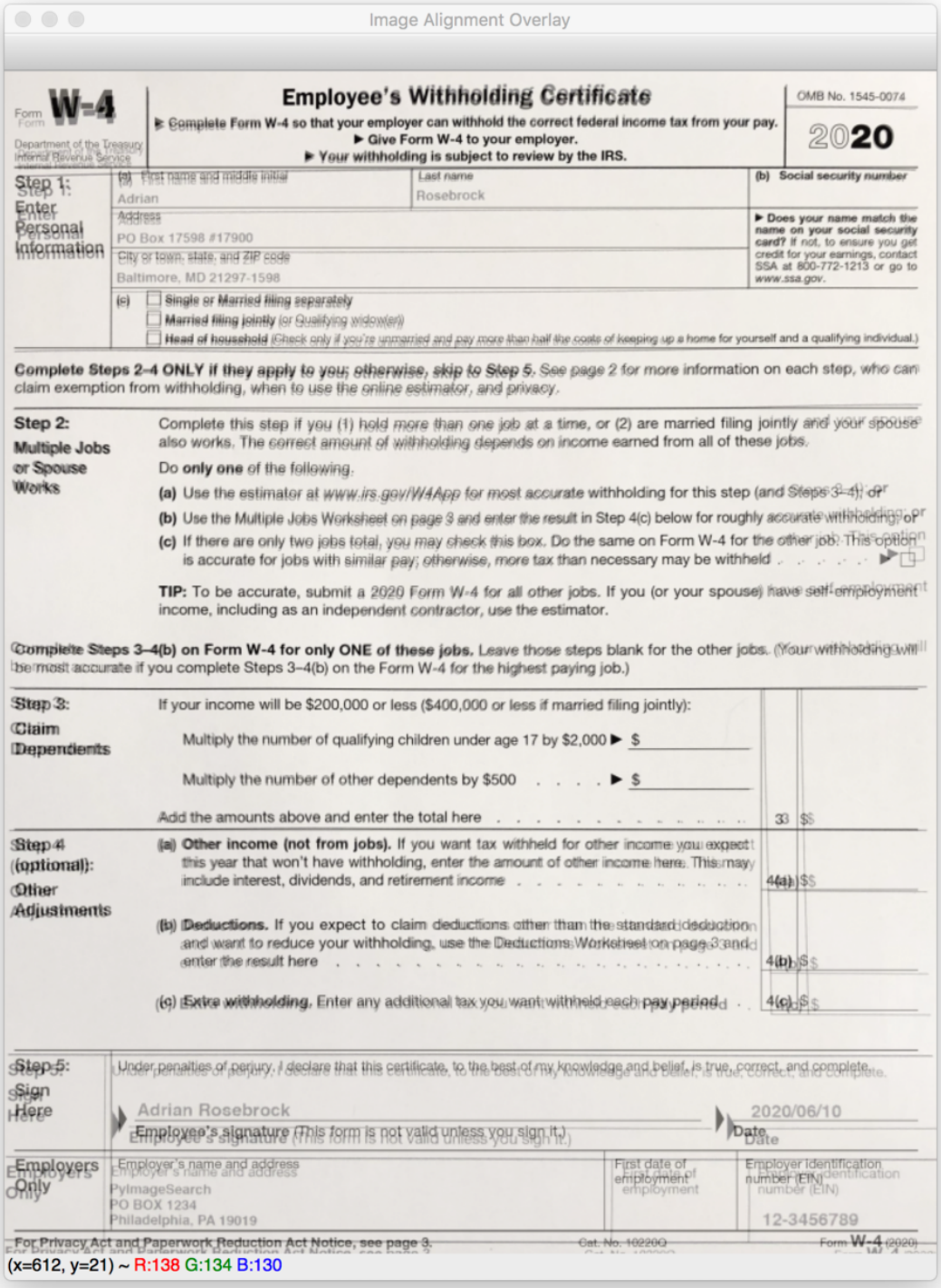


Figure 6.6. This time, we overlay our aligned and registered image on our template with a 50/50 blend. Our overlay method makes the differences pop. Notice there is a slight effect of "double vision" where there are some minor differences.

Our alignment isn't *perfect* (obtaining a pixel-perfect alignment is incredibly challenging and, in some cases, unrealistic). Still, the form's fields are sufficiently aligned such that we'll be able to OCR the text and associate the fields together.

6.3 Summary

In this chapter, you learned how to perform image alignment and registration using OpenCV.

Image alignment has many use cases, including medical scans, military-based automatic target acquisition, and satellite image analysis.

We chose to examine image alignment for one of the most important (and most utilized) purposes — **optical character recognition (OCR).**

Applying image alignment to an input image allows us to align it with a template document. Once we have the input image aligned with the template, we can apply OCR to recognize the text in each field. And since we know the location of each of the document fields, it becomes easy to associate the OCR'd text with each field.

Chapter 7

OCR'ing a Document, Form, or Invoice

In this chapter, you will learn how to OCR a document, form, or invoice using Tesseract, OpenCV, and Python.

Previously, we discovered how to accept an input image and align it to a template image. Recall that we had a template image — in our case, it was a form from the U.S. Internal Revenue Service. We also had the image that we wished to align with the template, allowing us to match fields. The process resulted in the alignment of both the template and the form for OCR.

We just need to associate fields on the form such as name, address, EIN, and more. Knowing where and what the fields are allows us to OCR each field and keep track of them for further processing (e.g., automated database entry). However, that raises the questions:

- How do we go about implementing this document OCR pipeline?
- What OCR algorithms will we need to utilize?
- And how *complicated* is this OCR application going to be?
- And most importantly, can we exercise our computer vision and OCR skills to implement our entire document OCR pipeline in under 150 lines of code?

Of course, I wouldn't ask such a question unless the answer is "yes." Here's why we are spending so much time OCR'ing forms. Forms are everywhere.

Schools have many forms for application and enrollment. When's the last time you've gone through the scholarship process?

Healthcare is also full of forms. Both the doctor and the dentist are likely to ask you to fill out a form at your next visit.

We also get forms for property like real estate and homeowners insurance and don't forget about business forms.

If you want to vote, get a permit, or get a passport, you are in for more forms.

As you can see, forms are everywhere so we want to learn to OCR them, let's dive in!

7.1 Automatically Registering and OCR'ing a Form

In the first part of this chapter, we'll briefly discuss why we may want to OCR documents, forms, invoices, or any type of physical document. From there, we'll review the steps required to implement a document OCR pipeline. We'll then implement each of the individual steps in a Python script using OpenCV and Tesseract. Finally, we'll review the results of applying image alignment and OCR to our sample images.

7.1.1 Why Use OCR on Forms, Invoices, and Documents?

Despite living in the digital age, we still have a strong reliance on *physical* paper trails, *especially* in large organizations such as government, enterprise companies, and universities/colleges. The need for physical paper trails combined with the fact that nearly every document needs to be organized, categorized, and even *shared* with multiple people in an organization requires that we also *digitize* the information on the document and store it in our databases. Large organizations employ data entry teams whose sole purpose is to take these physical documents, manually re-type the information, and then save it into the system.

Optical character recognition (OCR) algorithms can *automatically* digitize these documents, extract the information, and pipe them into a database for storage, alleviating the need for large, expensive, and even error-prone manual entry teams. In this chapter, you'll learn how to implement a basic document OCR pipeline using OpenCV and Tesseract.

7.1.2 Steps to Implementing a Document OCR Pipeline with OpenCV and Tesseract

Implementing a document OCR pipeline with OpenCV and Tesseract is a multi-step process. In this section, we'll discover the **five steps** required for creating a pipeline to OCR a form.

Step #1 involves defining the locations of fields in the input image document. We can do this by opening our template image in our favorite image editing software, such as Photoshop, GNU Image Manipulation Program (GIMP), or whatever photo application is built into your operating system. From there, we manually examine the image and determine the bounding box (*x, y*)-coordinates of each field we want to OCR, as shown on the *top* of Figure 7.1.

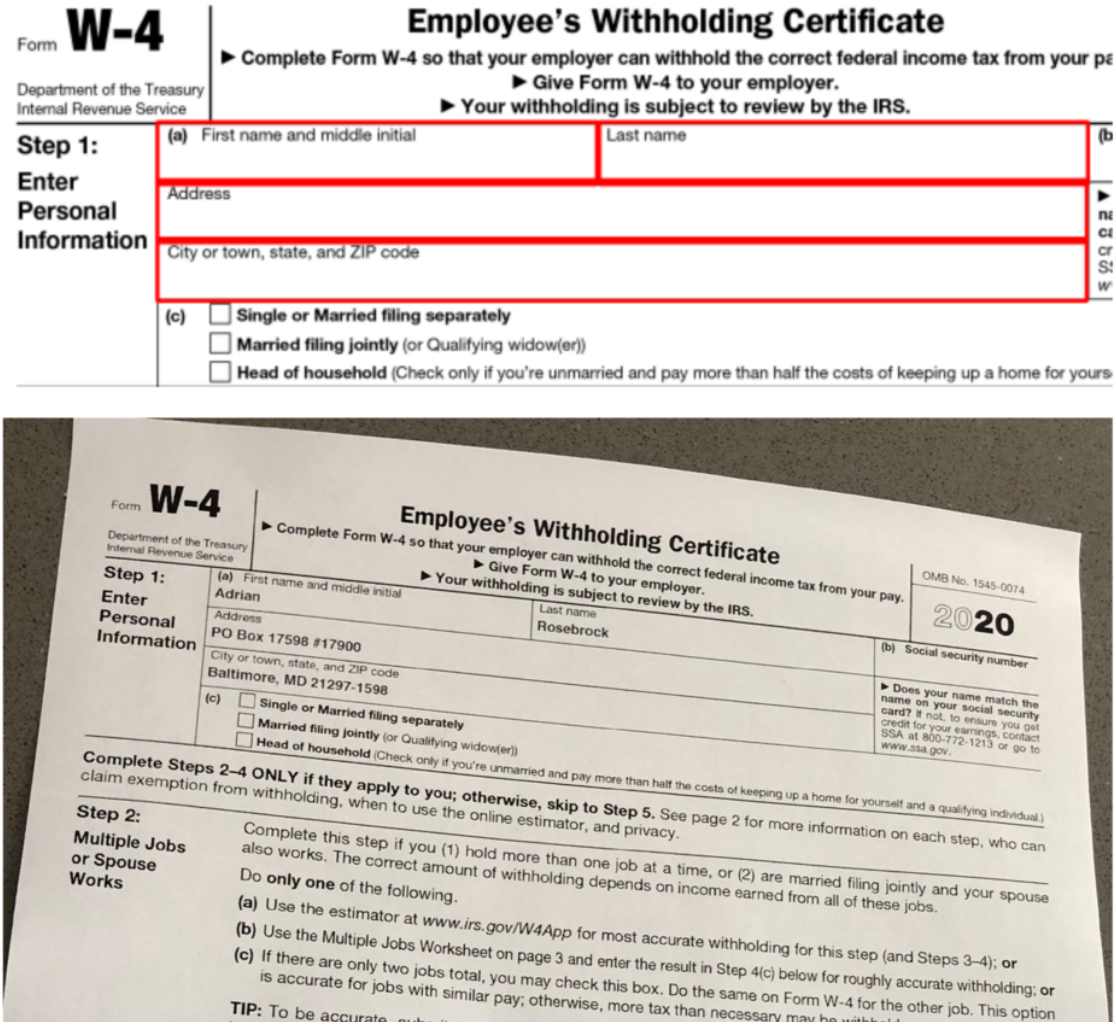


Figure 7.1. *Top:* Specifying the locations in a document (i.e., form fields) is **Step #1** in implementing a document OCR pipeline with OpenCV, Tesseract, and Python. *Bottom:* Presenting an image (e.g., a document scan or smartphone photo of a document on a desk) to our OCR pipeline is **Step #2** in our automated OCR system based on OpenCV, Tesseract, and Python.

Then, in **Step #2**, we accept an input image containing the document we want to OCR and present it to our OCR pipeline. Figure 7.1 shows both the image *top-left* and the original form *top-right*.

We can then (**Step #3**) apply automatic image alignment/registration to align the input image with the template form (Figure 7.2, *top-left*). In **Step #4**, we loop over all text field locations (which we defined in **Step #1**), extract the ROI, and apply OCR to the ROI. During this step, we're able to OCR the text itself *and* associate it with a text field in the original template document.

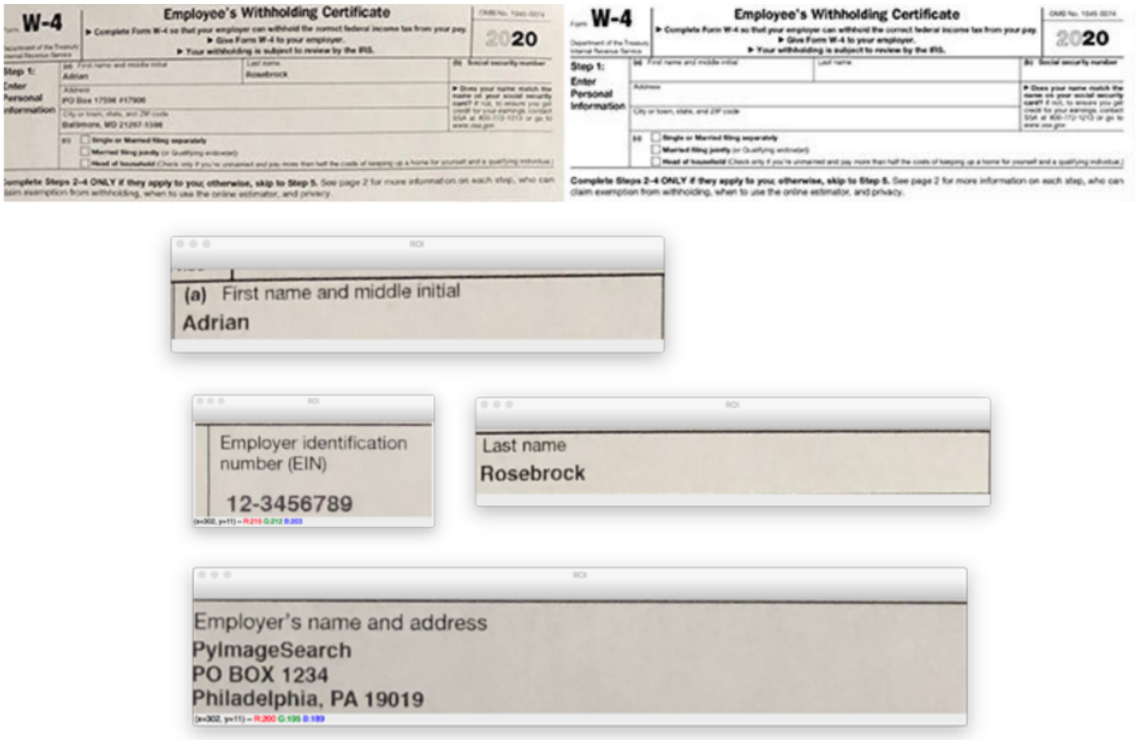


Figure 7.2. *Top-left:* Aligning a scanned document with its template using OpenCV and Python represents **Step #3** of our OCR pipeline. *Top-right, Middle, and Bottom:* Knowing the form field locations from **Step #1** allows us to perform **Step #4**, which consists of extracting ROIs from our aligned document and accomplishing OCR.

The final **Step #5** is to display our output OCR'd document depicted in Figure 7.3.

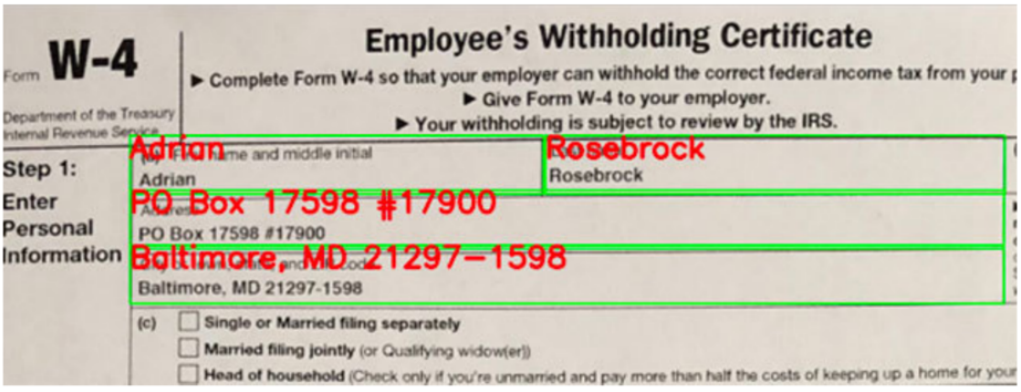


Figure 7.3. Finally, **Step #5** in our OCR pipeline takes action with the OCR'd text data. Given that this is a proof of concept, we'll simply annotate the OCR'd text data on the aligned scan for verification. This is the point where a real-world system would pipe the information into a database or make a decision based upon it (e.g., perhaps you need to apply a mathematical formula to several fields in your document).

We'll learn how to develop a Python script to accomplish **Steps #1–#5** in this chapter by creating an OCR document pipeline using OpenCV and Tesseract.

7.1.3 Project Structure

```
|-- pyimagesearch
|   |-- alignment
|   |   |-- __init__.py
|   |   |-- align_images.py
|   |-- __init__.py
|   |-- helpers.py
|-- scans
|   |-- scan_01.jpg
|   |-- scan_02.jpg
|-- form_w4.png
|-- ocr_form.py
```

The directory and file structure for this chapter are very straightforward. We have three images:

- `scans/scan_01.jpg`: A sample IRS W-4 has been filled with fake tax information, including my name.
- `scans/scan_02.jpg`: A similar sample IRS W-4 document that has been populated with fake tax information.
- `form_w4.png`: The official 2020 IRS W-4 form template. This empty form does not have any information entered into it. We need it and the field locations to line up the scans and ultimately extract information from the scans. We'll manually determine the field locations with an external photo editing/previewing application.

And we have just a single Python driver script to review: `ocr_form.py`. This form parser relies on two helper functions in the `pyimagesearch` module that we're familiar with from previous chapters:

- `align_images`: Contained within the `alignment` submodule
- `cleanup_text`: In our `helpers.py` file

If you're ready to dive in, simply head to the implementation section next!

7.1.4 Implementing Our Document OCR Script with OpenCV and Tesseract

We are now ready to implement our document OCR Python script using OpenCV and Tesseract. Open a new file, name it `ocr_form.py`, and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.alignment import align_images
3  from pyimagesearch.helpers import cleanup_text
4  from collections import namedtuple
5  import pytesseract
6  import argparse
7  import imutils
8  import cv2
```

You should recognize each of the imports on **Lines 2–8**; however, let's highlight a few of them. The `cleanup_text` helper function is used to strip out non-ASCII text from a string. We need to cleanse our text because OpenCV's `cv2.putText` is unable to draw non-ASCII characters on an image (unfortunately, OpenCV replaces each unknown character with a `?`). Of course, our effort is a lot easier when we use OpenCV, PyTesseract, and `imutils`.

Next, let's handle our command line arguments:

```
10 # construct the argument parser and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-i", "--image", required=True,
13     help="path to input image that we'll align to template")
14 ap.add_argument("-t", "--template", required=True,
15     help="path to input template image")
16 args = vars(ap.parse_args())
```

Our script requires two command line arguments: (1) `--image` is our input image of a form or invoice and (2) `--template` is the path to our template form or invoice. We'll align our image to the template and then OCR various fields as needed.

We aren't creating a "smart form OCR system" in which all text is recognized and fields are designed based on regular expression patterns. That is certainly doable, but to keep this chapter lightweight, I've manually defined `OCR_Locations` for each field about which we are concerned. The benefit is that we'll be able to give each field a name and specify the exact (*x, y*)-coordinates serving as the bounds of the field. Let's work on defining the text field locations in **Step #1** now:

```
18 # create a named tuple which we can use to create locations of the
19 # input document which we wish to OCR
20 OCRLocation = namedtuple("OCRLocation", ["id", "bbox",
21     "filter_keywords"])
22
23 # define the locations of each area of the document we wish to OCR
24 OCR_LOCATIONS = [
25     OCRLocation("step1_first_name", (265, 237, 751, 106),
26         ["middle", "initial", "first", "name"]),
```

```
27      OCRLocation("step1_last_name", (1020, 237, 835, 106),
28          ["last", "name"]),
29      OCRLocation("step1_address", (265, 336, 1588, 106),
30          ["address"]),
31      OCRLocation("step1_city_state_zip", (265, 436, 1588, 106),
32          ["city", "zip", "town", "state"]),
33      OCRLocation("step5_employee_signature", (319, 2516, 1487, 156),
34          ["employee", "signature", "form", "valid", "unless",
35              "you", "sign"]),
36      OCRLocation("step5_date", (1804, 2516, 504, 156), ["date"]),
37      OCRLocation("employee_name_address", (265, 2706, 1224, 180),
38          ["employer", "name", "address"]),
39      OCRLocation("employee_ein", (1831, 2706, 448, 180),
40          ["employer", "identification", "number", "ein"]),
41  ]
```

Here, **Lines 20 and 21** create a named tuple consisting of the following:

- `"OCRLocation"`: The name of our tuple.
- `"id"`: A short description of the field for easy reference. Use this field to describe what the form field is. For example, is it a zip-code field?
- `"bbox"`: The bounding box coordinates of a field in list form using the following order: `[x, y, w, h]`. In this case, *x* and *y* are the *top-left* coordinates, and *w* and *h* are the width and height.
- `"filter_keywords"`: A list of words that we do not wish to consider for OCR, such as form field instructions, as demonstrated in Figure 7.4.

Figure 7.4. A W-4 form with eight fields outlined in red.

Lines 24–41 define ***eight* fields of an official 2020 IRS W-4 tax form**, pictured in Figure 7.4. Bounding box coordinates (`"bbox"`) were manually determined by inspecting the (*x, y*)-coordinates of the image. This can be accomplished using any photo editing application, including Photoshop, GIMP, or the basic preview/paint application built into your operating system. Alternatively, you could use OpenCV mouse click events per my blog post, *Capturing Mouse Click Events with Python and OpenCV* (http://pyimg.co/fmckl [24]).

Now that we've handled imports, configured command line arguments, and defined our OCR field locations, let's go ahead and **load and align** our input `--image` to our `--template` (**Step #2 and Step #3**):

```
43 # load the input image and template from disk
44 print("[INFO] loading images...")
45 image = cv2.imread(args["image"])
46 template = cv2.imread(args["template"])
47
48 # align the images
49 print("[INFO] aligning images...")
50 aligned = align_images(image, template)
```

As you can see, **Lines 45 and 46** load both our input `--image` (e.g., a scan or snap from your smartphone camera) and our `--template` (e.g., a document straight from the IRS, your mortgage company, accounting department, or anywhere else depending on your needs).

Remark 7.1. *You may be wondering how I converted the `form_w4.png` from PDF form (most IRS documents are PDFs these days). This process is straightforward with a free OS-agnostic tool called ImageMagick (Figure 7.5). With ImageMagick installed, you can simply use the `convert` command (http://pyimg.co/ekwbu [25]). For example, you could enter the following command:*

```
$ convert /path/to/taxes/2020/forms/form_w4.pdf ./form_w4.png
```

As you can see, ImageMagick is smart enough to recognize that you want to convert a PDF to a PNG image based on a combination of the file extension as well as the file itself. You could alter the command quite easily to produce a JPG if you'd like. Once your form is in PNG or JPG form, you can use it with OpenCV and PyTesseract! Do you have a lot of forms? Simply use the `mogrify` command, which supports wildcard operators (http://pyimg.co/t16bj [26]).

Once the image files are loaded into memory, we simply take advantage of our `align_images` helper utility (**Line 50**) to perform the alignment. Figure 7.6 shows the result of aligning our `scan01.jpg` input to the form template. Notice how our input image *(left)* has been aligned to the template document *(right)*.

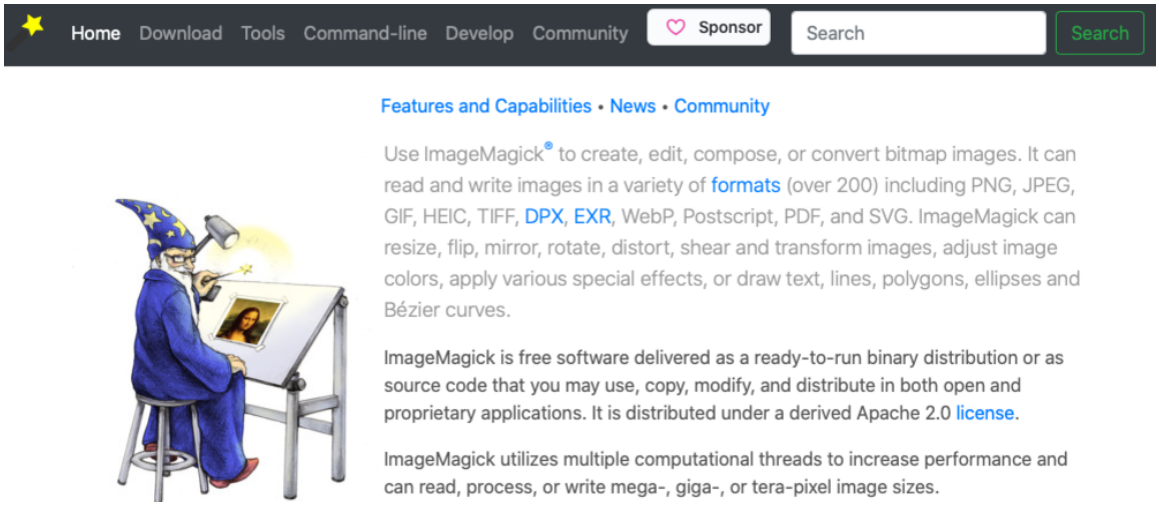


Figure 7.5. ImageMagick software webpage.

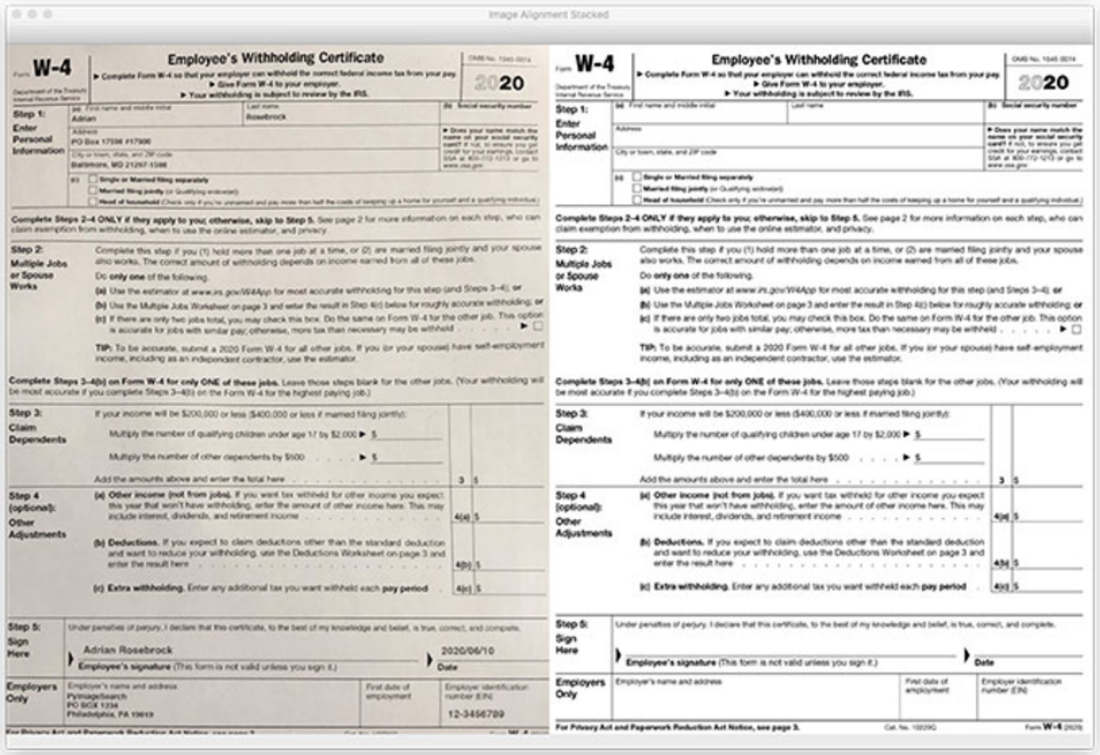


Figure 7.6. *Left:* an image of a W-4 form. *Right:* a W-4 Form.

The next step (**Step #4**) is to loop over each of our `OCR_LOCATIONS` and **apply OCR to each of the text fields** using the power of Tesseract and PyTesseract:

```
52 # initialize a results list to store the document OCR parsing results
53 print("[INFO] OCR'ing document...")
```

```
54 parsingResults = []
55
56 # loop over the locations of the document we are going to OCR
57 for loc in OCR_LOCATIONS:
58     # extract the OCR ROI from the aligned image
59     (x, y, w, h) = loc.bbox
60     roi = aligned[y:y + h, x:x + w]
61
62     # OCR the ROI using Tesseract
63     rgb = cv2.cvtColor(roi, cv2.COLOR_BGR2RGB)
64     text = pytesseract.image_to_string(rgb)
```

First, we initialize the `parsingResults` list to store our OCR results for each field of text (**Line 54**). From there, we proceed to loop over each of the `OCR_LOCATIONS` (beginning on **Line 57**), which we have previously manually defined.

Inside the loop (**Lines 59–64**), we begin by (1) extracting the particular text field ROI from the aligned image and (2) use PyTesseract to OCR the ROI. Remember, Tesseract expects an RGB format image, so **Line 63** swaps color channels accordingly.

Now let's break each OCR'd `text` field into individual lines/rows:

```
66     # break the text into lines and loop over them
67     for line in text.split("\n"):
68         # if the line is empty, ignore it
69         if len(line) == 0:
70             continue
71
72         # convert the line to lowercase and then check to see if the
73         # line contains any of the filter keywords (these keywords
74         # are part of the *form itself* and should be ignored)
75         lower = line.lower()
76         count = sum([lower.count(x) for x in loc.filter_keywords])
77
78         # if the count is zero then we know we are *not* examining a
79         # text field that is part of the document itself (ex., info,
80         # on the field, an example, help text, etc.)
81         if count == 0:
82             # update our parsing results dictionary with the OCR'd
83             # text if the line is *not* empty
84             parsingResults.append((loc, line))
```

Line 67 begins a loop over the `text` lines where we immediately ignore empty lines (**Lines 69 and 70**). Assuming the `line` isn't empty, we filter it for keywords (forcing to lower-case characters in the process) to ensure that we aren't examining a part of the document itself. In other words, we only care about form-filled information and not the instructional text on the template form itself. **Lines 75–84** accomplish the filtering process and add the OCR'd field to `parsingResults` accordingly.

For example, consider the *"First name and middle initial"* field in Figure 7.7. While I've filled out this field with my first name, *"Adrian,"* the text *"(a) First name and middle initial"* will still be OCR'd by Tesseract — the code above *automatically filters out* the instructional text inside the field, ensuring only the human inputted text is returned.

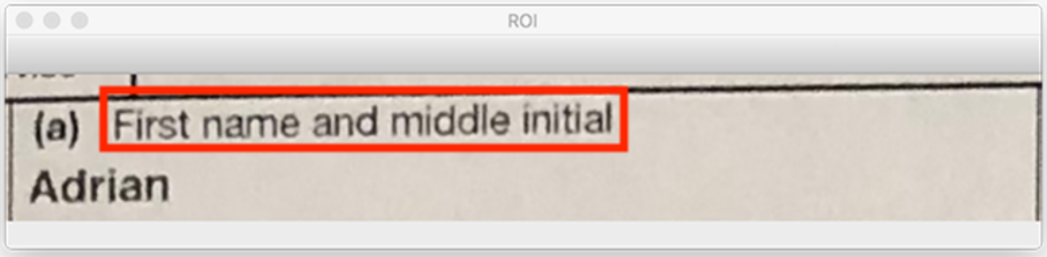


Figure 7.7. First and middle initial field filled in with Adrian

I hope you're still with me. Let's carry on by post-processing our `parsingResults` to clean them up:

```
86  # initialize a dictionary to store our final OCR results
87  results = {}
88
89  # loop over the results of parsing the document
90  for (loc, line) in parsingResults:
91      # grab any existing OCR result for the current ID of the document
92      r = results.get(loc.id, None)
93
94      # if the result is None, initialize it using the text and location
95      # namedtuple (converting it to a dictionary as namedtuples are not
96      # hashable)
97      if r is None:
98          results[loc.id] = (line, loc._asdict())
99
100     # otherwise, there exists a OCR result for the current area of the
101     # document, so we should append our existing line
102     else:
103         # unpack the existing OCR result and append the line to the
104         # existing text
105         (existingText, loc) = r
106         text = "{}\n{}".format(existingText, line)
107
108         # update our results dictionary
109         results[loc["id"]] = (text, loc)
```

Our final `results` dictionary (**Line 87**) will soon hold the cleansed parsing results consisting of the unique ID of the text location (key) and the 2-tuple of the OCR'd text and its location

(value). Let's begin populating our `results` by looping over our `parsingResults` on **Line 90**. Our loop accomplishes three tasks:

- We grab any existing result for the current text field ID
- If there is no current result, we simply store the text `line` and text `loc` (location) in the `results` dictionary
- Otherwise, we *append the* `line` to any *`existingText`* separated by a newline for the field and update the `results` dictionary

We're finally ready to perform **Step #5** — visualizing our OCR `results`:

```
111 # loop over the results
112 for (locID, result) in results.items():
113     # unpack the result tuple
114     (text, loc) = result
115
116     # display the OCR result to our terminal
117     print(loc["id"])
118     print("=" * len(loc["id"]))
119     print("{}\n\n".format(text))
120
121     # extract the bounding box coordinates of the OCR location and
122     # then strip out non-ASCII text so we can draw the text on the
123     # output image using OpenCV
124     (x, y, w, h) = loc["bbox"]
125     clean = cleanup_text(text)
126
127     # draw a bounding box around the text
128     cv2.rectangle(aligned, (x, y), (x + w, y + h), (0, 255, 0), 2)
129
130     # loop over all lines in the text
131     for (i, line) in enumerate(text.split("\n")):
132         # draw the line on the output image
133         startY = y + (i * 70) + 40
134         cv2.putText(aligned, line, (x, startY),
135             cv2.FONT_HERSHEY_SIMPLEX, 1.8, (0, 0, 255), 5)
```

Looping over each of our `results` begins on **Line 112**. Our first task is to unpack the 2-tuple consisting of the OCR'd and parsed `text` as well as its `loc` (location) via **Line 114**. Both of these results are then printed in our terminal (**Lines 117–119**).

From there, we extract the bounding box coordinates of the text field (**Line 124**). Subsequently, we strip out non-ASCII characters from the OCR'd `text` via our `cleanup_text` helper utility (**Line 125**). Cleaning up our `text` ensures we can use OpenCV's `cv2.putText` function to annotate the output image.

We then proceed to draw the bounding box rectangle around the `text` (**Line 128**) and annotate each `line` of `text` (delimited by newlines) on the output image (**Lines 131–135**).

Finally, we'll display our (1) original input `--image` and (2) annotated output result:

```
137 # show the input and output images, resizing it such that they fit
138 # on our screen
139 cv2.imshow("Input", imutils.resize(image, width=700))
140 cv2.imshow("Output", imutils.resize(aligned, width=700))
141 cv2.waitKey(0)
```

As you can see, our `cv2.imshow` commands also apply aspect-aware resizing because high resolution scans tend not to fit on the average computer screen (**Lines 139 and 140**). To stop the program, simply press any key while one of the windows is in focus.

Great job implementing your automated OCR system with Python, OpenCV, and Tesseract! In the next section, we'll put it to the test.

7.1.5 OCR Results Using OpenCV and Tesseract

We are now ready to OCR our document using OpenCV and Tesseract. Fire up your terminal, find the chapter directory, and enter the following command:

```
$ python ocr_form.py --image scans/scan_01.jpg --template form_w4.png
[INFO] loading images...
[INFO] aligning images...
[INFO] OCR'ing document...
step1_first_name
================
Adrian

step1_last_name
===============
Rosebrock

step1_address
=============
PO Box 17598 #17900

step1_city_state_zip
====================
Baltimore, MD 21297-1598

step5_employee_signature
========================
Adrian Rosebrock
```

```
step5_date
==========
2020/06/10

employee_name_address
=====================
PylmageSearch
PO BOX 1234
Philadelphia, PA 19019

employee_ein
============
12-3456789
```

As Figure 7.8 demonstrates, we have our input image and its corresponding template. And in Figure 7.9, we have the output of the image alignment and document the OCR pipeline. Notice how we've successfully aligned our input image with the document template, localize each of the fields, and then OCR each of the fields.

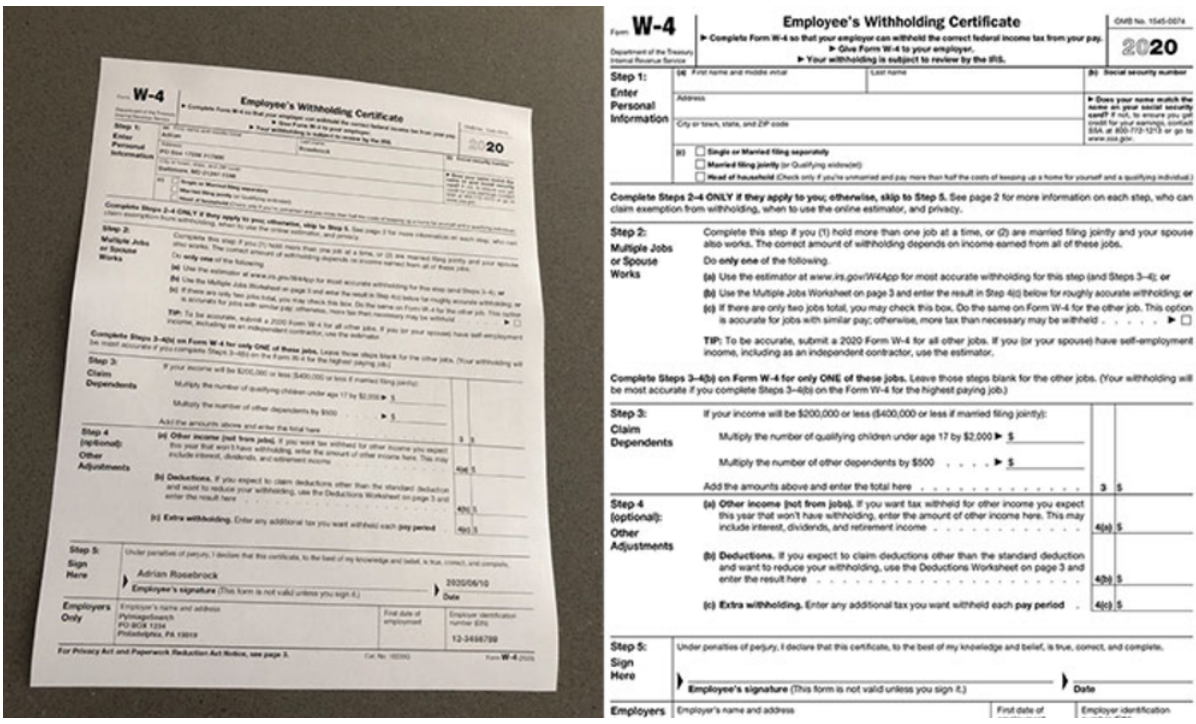


Figure 7.8. *Left:* an input image. *Right:* a corresponding template.

Beware that our pipeline *ignores* any line of text inside a field *that is part of the document itself.* For example, the first name field provides the instructional text *"(a) First name and middle initial."* However, our OCR pipeline and keyword filtering process can detect that this is

part of the document itself (i.e., not something a *human* entered) and then simply ignores it. Overall, we've been able to successfully OCR the document!

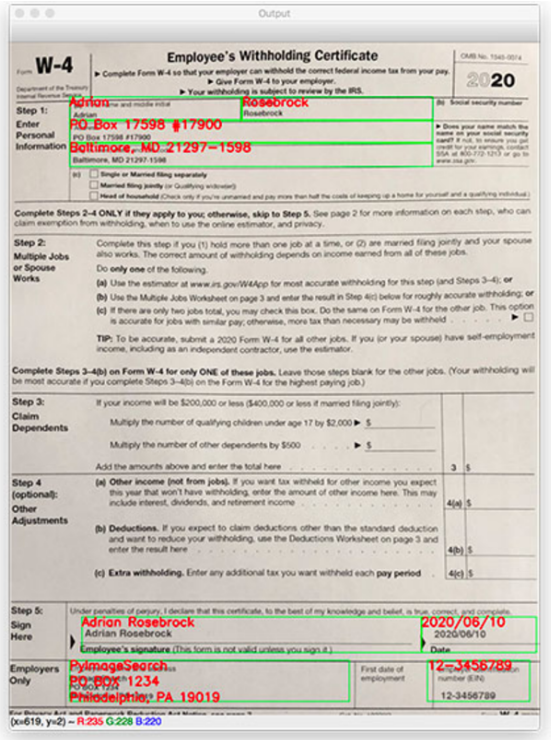


Figure 7.9. Output of the image alignment and document OCR pipeline.

Let's try another sample image, this time with a slightly different viewing angle:

```
$ python ocr_form.py --image scans/scan_02.jpg --template form_w4.png
[INFO] loading images...
[INFO] aligning images...
[INFO] OCR'ing document...
step1_first_name
================
Adrian

step1_last_name
===============
Rosebrock

step1_address
=============
PO Box 17598 #17900

step1_city_state_zip
====================
Baltimore, MD 21297-1598
```

```
step5_employee_signature
========================
Adrian Rosebrock

step5_date
==========
2020/06/10

employee_name_address
=====================
PyimageSearch
PO BOX 1234
Philadelphia, PA 19019

employee_ein
============
12-3456789
```

Figure 7.10 shows our input image along with its template. Figure 7.11 contains our output, where you can see that the image is aligned with the template and the OCR successfully applied to each of the fields. As you can see, we've successfully aligned the input image with the template document and then OCR each of the individual fields!

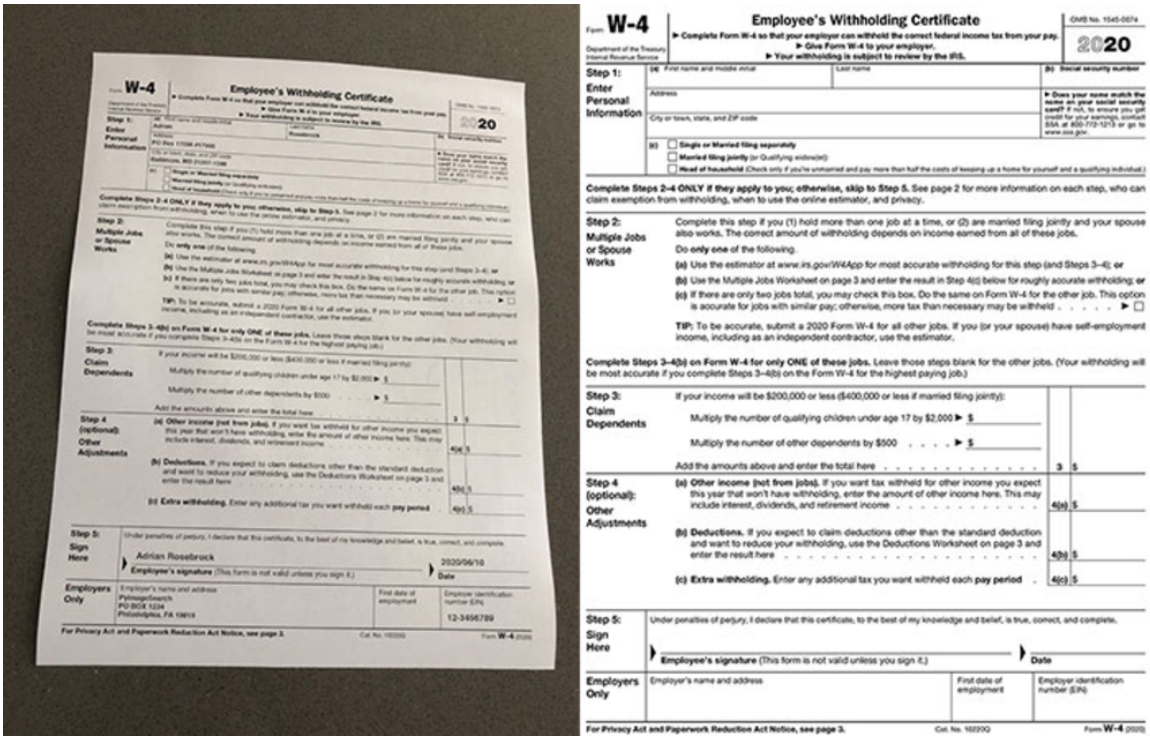


Figure 7.10. *Left:* an input image. *Right:* a corresponding template.

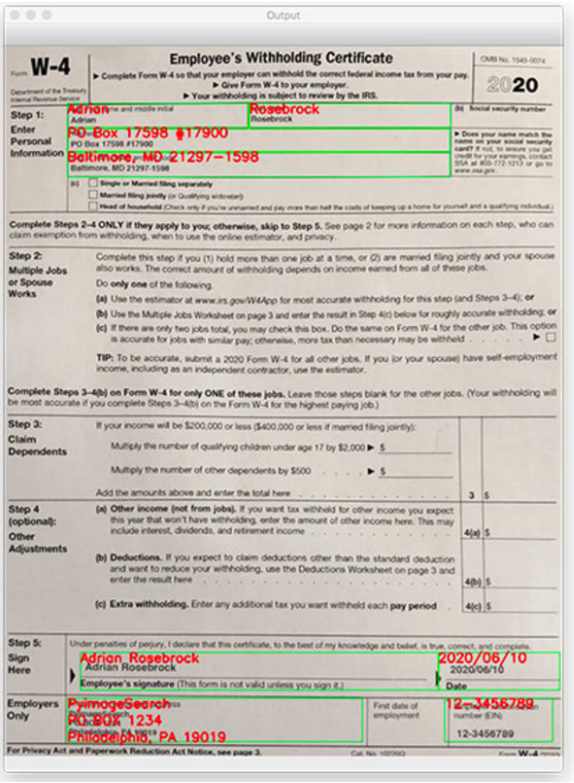


Figure 7.11. Output of the image alignment and document OCR pipeline.

7.2 Summary

In this chapter, you learned how to OCR a document, form, or invoice using OpenCV and Tesseract.

Our method hinges on image alignment which we covered earlier in this book, which is the process of accepting an input image and a template image, and then aligning them such that they can neatly "overlay" on top of each other. Image alignment allows us to align each of the text fields in a template with our input image, meaning that once we've OCR'd the document, we can associate the OCR'd text to each field (e.g., name, address, etc.).

Once image alignment was applied, we used Tesseract to localize text in the input image. Tesseract's text localization procedure gave us the (x, y)-coordinates of each text location, which we then mapped to the appropriate text field in the template. In this manner, we scanned and recognized each important text field of the input document!

Chapter 10

OCR'ing Business Cards

In our previous chapter, we learned how to automatically OCR and scan receipts by:

i. Detecting the receipt in the input image
ii. Applying a perspective transform to obtain a *top-down* view of the receipt
iii. Utilizing Tesseract to OCR the text on the receipt
iv. Applying regular expressions to extract the price data

In this chapter, we're going to use a very similar workflow, but this time apply it to business card OCR. **More specifically, we'll learn how to extract the name, title, phone number, and email address from a business card.**

You'll then be able to extend this implementation to your projects.

10.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how to detect business cards in images
ii. Apply OCR to a business card image
iii. Utilize regular expressions to extract:
    a. Name
    b. Job title
    c. Phone number
    d. Email address

10.2 Business Card OCR

In the first part of this chapter, we will review our project directory structure. We'll then implement a simple yet effective Python script to allow us to OCR a business card.

We'll wrap up this chapter with a discussion of our results, along with the next steps.

10.2.1 Project Structure

Before we get too far, let's take a look at our project directory structure:

```
|-- larry_page.png
|-- ocr_business_card.py
|-- tony_stark.png
```

We only have a single Python script to review, `ocr_business_card.py`. This script will load example business card images (i.e., `larry_page.png` and `tony_stark.png`), OCR them, and then output the name, job title, phone number, and email address from the business card.

Best of all, we'll be able to accomplish our goal in under 120 lines of code (including comments)!

10.2.2 Implementing Business Card OCR

We are now ready to implement our business card OCR script! Open the `ocr_business_card.py` file in our project directory structure and insert the following code:

```
1 # import the necessary packages
2 from imutils.perspective import four_point_transform
3 import pytesseract
4 import argparse
5 import imutils
6 import cv2
7 import re
```

Our imports here are similar to the ones in our previous chapter on OCR'ing receipts.

We need our `four_point_transform` function to obtain a *top-down*, bird's-eye view of the business card. Obtaining this view typically yields higher OCR accuracy.

The `pytesseract` package is used to interface with the Tesseract OCR engine. We then have Python's regular expression library, `re`, which will allow us to parse the names, job titles, email addresses, and phone numbers from business cards.

With the imports taken care of, we can move on to command line arguments:

```
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
    help="path to input image")
ap.add_argument("-d", "--debug", type=int, default=-1,
    help="whether or not we are visualizing each stpe of the pipeline")
ap.add_argument("-c", "--min-conf", type=int, default=0,
    help="minimum confidence value to filter weak text detection")
args = vars(ap.parse_args())
```

Our first command line argument, `--image`, is the path to our input image on disk. We assume that this image contains a business card with sufficient contrast between the foreground and background, ensuring we can apply edge detection and contour processing to successfully extract the business card.

We then have two optional command line arguments, `--debug` and `--min-conf`. The `--debug` command line argument is used to indicate if we are debugging our image processing pipeline and showing more of the processed images on our screen (useful for when you can't determine why a business card was detected or not).

We then have `--min-conf`, which is the minimum confidence (on a scale of 0–100) required for successful text detection. You can increase `--min-conf` to prune out weak text detections.

Let's now load our input image from disk:

```
# load the input image from disk, resize it, and compute the ratio
# of the *new* width to the *old* width
orig = cv2.imread(args["image"])
image = orig.copy()
image = imutils.resize(image, width=600)
ratio = orig.shape[1] / float(image.shape[1])
```

Here we load our input `--image` from disk and then clone it. We make it a clone so that we can extract the original high-resolution version of the business card after contour processing.

We then resize our `image` to have a width of 600px and then compute the ratio of the *new* width to the *old* width (a requirement for when we want to obtain a *top-down* view of the original high-resolution business card).

We continue our image processing pipeline below.

```
# convert the image to grayscale, blur it, and apply edge detection
# to reveal the outline of the business card
```

```
28  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
29  blurred = cv2.GaussianBlur(gray, (5, 5), 0)
30  edged = cv2.Canny(blurred, 30, 150)
31
32  # detect contours in the edge map, sort them by size (in descending
33  # order), and grab the largest contours
34  cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
35      cv2.CHAIN_APPROX_SIMPLE)
36  cnts = imutils.grab_contours(cnts)
37  cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:5]
38
39  # initialize a contour that corresponds to the business card outline
40  cardCnt = None
```

First, we take our original `image` and then convert it to grayscale, blur it, and then apply edge detection, the result of which can be seen in Figure 10.1.

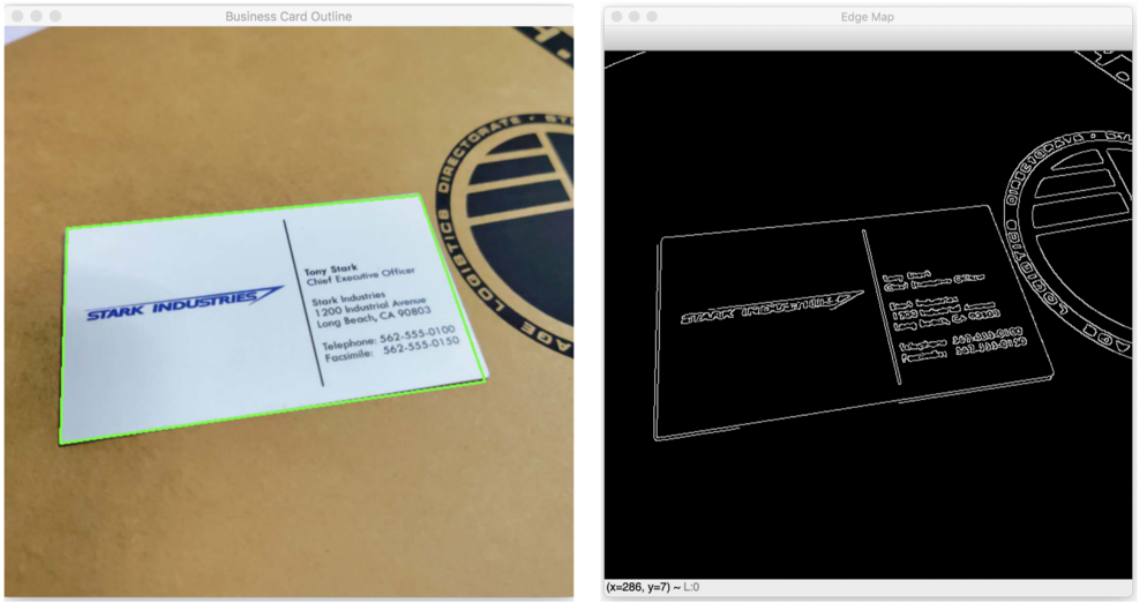


Figure 10.1. *Left:* Our input image containing a business card. *Right:* Computing the edge map of the input image reveals the outlines of the business card.

Note that the outline/border of the business card is visible on the edge map. Suppose there are any gaps in the edge map. In that case, the business card will *not* be detectable via our contour processing technique, so you may need to tweak the parameters to the Canny edge detector or capture your image in an environment with better lighting conditions.

From there, we detect contours and sort them in descending order (largest to smallest) based on the area of the computed contour. Our assumption here will be the business card contour will be one of the largest detected contours, hence this operation.

We also initialize `cardCnt` (**Line 40**), which is the contour that corresponds to the business card.

Let's now loop over the largest contours:

```
42  # loop over the contours
43  for c in cnts:
44      # approximate the contour
45      peri = cv2.arcLength(c, True)
46      approx = cv2.approxPolyDP(c, 0.02 * peri, True)
47
48      # if this is the first contour we've encountered that has four
49      # vertices, then we can assume we've found the business card
50      if len(approx) == 4:
51          cardCnt = approx
52          break
53
54  # if the business card contour is empty then our script could not
55  # find the  outline of the card, so raise an error
56  if cardCnt is None:
57      raise Exception(("Could not find receipt outline. "
58          "Try debugging your edge detection and contour steps."))
```

Lines 45 and 46 perform contour approximation.

If our approximated contour has four vertices, then we can assume that we found the business card. If that happens, we `break` from the loop and update our `cardCnt`.

If we reach the end of the `for` loop and still haven't found a valid `cardCnt`, then we gracefully exit the script. Remember, we cannot process the business card if one cannot be found in the image!

Our next code block handles showing some debugging images as well as obtaining our *top-down* view of the business card:

```
60  # check to see if we should draw the contour of the business card
61  # on the image and then display it to our screen
62  if args["debug"] > 0:
63      output = image.copy()
64      cv2.drawContours(output, [cardCnt], -1, (0, 255, 0), 2)
65      cv2.imshow("Business Card Outline", output)
66      cv2.waitKey(0)
67
68  # apply a four-point perspective transform to the *original* image to
69  # obtain a top-down bird's-eye view of the business card
70  card = four_point_transform(orig, cardCnt.reshape(4, 2) * ratio)
71
72  # show transformed image
```

```
73  cv2.imshow("Business Card Transform", card)
74  cv2.waitKey(0)
```

Lines 62–66 make a check to see if we are in `--debug` mode, and if so, we draw the contour of the business card on the `output` image.

We then apply a four-point perspective transform to the *original, high-resolution image*, thus obtaining the *top-down*, bird's-eye view of the business card (**Line 70**).

We multiply the `cardCnt` by our computed `ratio` here since `cardCnt` was computed for the reduced image dimensions. Multiplying by `ratio` scales the `cardCnt` back into the dimensions of the `orig` image.

We then display the transformed image to our screen (**Lines 73 and 74**).

With our *top-down* view of the business card obtain, we can move on to OCR'ing it:

```
76  # convert the business card from BGR to RGB channel ordering and then
77  # OCR it
78  rgb = cv2.cvtColor(card, cv2.COLOR_BGR2RGB)
79  text = pytesseract.image_to_string(rgb)
80
81  # use regular expressions to parse out phone numbers and email
82  # addresses from the business card
83  phoneNums = re.findall(r'[\+\(]?[1-9][0-9 .\-\(\)]{8,}[0-9]', text)
84  emails = re.findall(r"[a-z0-9\.\-+_]+@[a-z0-9\.\-+_]+\.[a-z]+", text)
85
86  # attempt to use regular expressions to parse out names/titles (not
87  # necessarily reliable)
88  nameExp = r"^[\w'\-,.][^0-9_!¡?÷?¿/\\+=@#$%^&*(){}|~<>;:[\]]{2,}"
89  names = re.findall(nameExp, text)
```

Lines 78 and 79 OCR the business card, resulting in the `text` output.

But the question remains, how are we going to extract the information from the business card itself? **The answer is to utilize regular expressions.**

Lines 83 and 84 utilize regular expressions to extract phone numbers and email addresses (http://pyimg.co/eb5kf [40]) from the `text`, while **Lines 88 and 89** do the same for names and job titles (http://pyimg.co/oox3v [41]).

A review of regular expressions is outside the scope of this chapter, but the gist is that they can be used to match particular patterns in text.

For example, a phone number consists of a specific digits pattern, and sometimes includes dashes and parentheses. Email addresses also follow a pattern, including a string of text, followed by an "@" symbol, and then the domain name.

Any time you can reliably guarantee a pattern of text, regular expressions can work quite well. That said, they aren't perfect either, so you may want to look into more advanced natural language processing (NLP) algorithms if you find your business card OCR accuracy is suffering significantly.

The final step here is to display our output to the terminal:

```
91  # show the phone numbers header
92  print("PHONE NUMBERS")
93  print("==============")
94
95  # loop over the detected phone numbers and print them to our terminal
96  for num in phoneNums:
97      print(num.strip())
98
99  # show the email addresses header
100 print("\n")
101 print("EMAILS")
102 print("======")
103
104 # loop over the detected email addresses and print them to our
105 # terminal
106 for email in emails:
107     print(email.strip())
108
109 # show the name/job title header
110 print("\n")
111 print("NAME/JOB TITLE")
112 print("===============")
113
114 # loop over the detected name/job titles and print them to our
115 # terminal
116 for name in names:
117     print(name.strip())
```

This final code block loops over the extracted phone numbers (**Lines 96 and 97**), email addresses (**Lines 106 and 107**), and names/job titles (**Lines 116 and 117**), displaying each to our terminal.

Of course, you could take this extracted information, write to disk, save it to a database, etc. Still, for the sake of simplicity (and not knowing your project specifications of business card OCR), we'll leave it as an exercise to you to save the data as you see fit.

10.2.3 Business Card OCR Results

We are now ready to apply OCR to business cards. Open a terminal and execute the following command:

```
$ python ocr_business_card.py --image tony_stark.png --debug 1
PHONE NUMBERS
=============
562-555-0100
562-555-0150

EMAILS
======

NAME/JOB TITLE
==============
Tony Stark
Chief Executive Officer

Stark Industries
```

Figure 10.2 *(top)* shows the results of our business card localization. Notice how we have correctly detected the business card in the input image.

From there, Figure 10.2 *(bottom)* displays the results of applying a perspective transform of the business card, thus resulting in the *top-down*, bird's-eye view of the image.

Figure 10.2. *Top:* An input image containing Tony Stark's business card. *Bottom:* Obtaining a *top-down*, bird's-eye view of the input business card.

Once we have the *top-down* view of the image (typically required to obtain higher OCR accuracy), we can apply Tesseract to OCR it, the results of which can be seen in our terminal output above.

Note that our script has successfully extracted *both* phone numbers on Tony Stark's business card.

No email addresses are reported as there is no email address on the business card.

We then have the name and job title displayed as well. It's interesting to see that we are able to OCR all the text successfully because the text of the name is more distorted than the phone number text. Our perspective transform was able to deal with all the text effectively even though the amount of distortion changes as you go further away from the camera. That's the point of perspective transform and why it's important to the accuracy of our OCR.

Let's try another example image, this one of an old Larry Page (co-founder of Google) business card:

```
$ python ocr_business_card.py --image larry_page.png --debug 1
PHONE NUMBERS
=============
650 330-0100
650 618-1499

EMAILS
======
larry@google.com

NAME/JOB TITLE
==============
Larry Page
CEO

Google
```

Figure 10.3 *(top)* displays the output of localizing Page's business card. The *bottom* then shows the *top-down* transform of the image.

This *top-down* transform is passed through Tesseract OCR, yielding the OCR'd text as output. We take this OCR'd text, apply a regular expression, and thus obtain the results above.

Examining the results, you can see that we have successfully extracted Larry Page's two phone numbers, email address, and name/job title from the business card.

Figure 10.3. *Top:* An input image containing Larry Page's business card. *Bottom:* Obtaining a *top-down* view of the business card, which we can then OCR.

10.3 Summary

In this chapter, you learned how to build a basic business card OCR system. This system was, essentially, an extension of our receipt scanner, but with different regular expressions and text localization strategies.

If you ever need to build a business card OCR system, I recommend that you use this chapter as a starting point, but keep in mind that you may want to utilize more advanced text post-processing techniques, such as true natural language processing (NLP) algorithms, rather than regular expressions.

Regular expressions can work *very well* for email addresses and phone numbers, but for names and job titles that may fail to obtain high accuracy. If and when that time comes, you should consider leveraging NLP as much as possible to improve your results.

Chapter 17

Text Detection and OCR with Microsoft Cognitive Services

In our previous chapter, you learned how to use the Amazon Rekognition API to OCR images. The hardest part of using the Amazon Rekognition API was obtaining your API keys. Once you had your API keys, it was smooth sailing.

This chapter focuses on a different cloud-based API called Microsoft Cognitive Services (MCS), part of Microsoft Azure. Like Amazon Rekognition API, MCS is also capable of high OCR accuracy — but unfortunately, the implementation is slightly more complex (as is both Microsoft's login and admin dashboard).

I prefer the Amazon Web Services (AWS) Rekognition API over MCS, both for the admin dashboard and the API itself. However, if you are already ingrained into the MCS/Azure ecosystem, you should consider staying there. The MCS API isn't *that* hard to use (it's just not as easy and straightforward as Amazon Rekognition API).

17.1 Chapter Learning Objectives

In this chapter, you will:

- Learn how to obtain your MCS API keys
- Create a configuration file to store your subscription key and API endpoint URL
- Implement a Python script to make calls to the MCS OCR API
- Review the results of applying the MCS OCR API to sample images

17.2 Microsoft Cognitive Services for OCR

We'll start this chapter with a review of how you can obtain your MCS API keys. **You will need these API keys to make requests to the MCS API to OCR images.**

Once we have our API keys, we'll review our project directory structure and then implement a Python configuration file to store our subscription key and OCR API endpoint URL.

With our configuration file implemented, we'll move on to creating a second Python script, this one acting as a driver script that:

- Imports our configuration file
- Loads an input image to disk
- Packages the image into an API call
- Makes a request to the MCS OCR API
- Retrieves the results
- Annotates our output image
- Displays the OCR results to our screen and terminal

Let's dive in!

17.2.1 Obtaining Your Microsoft Cognitive Services Keys

Before proceeding to the rest of the sections, be sure to obtain the API keys by following the instructions shown here: pyimg.co/53rux.

17.2.2 Project Structure

The directory structure for our MCS OCR API is similar to the structure of the Amazon Rekognition API project in Chapter 16:

```
|-- config
|   |-- __init__.py
   |-- microsoft_cognitive_services.py
|-- images
|   |-- aircraft.png
|   |-- challenging.png
|   |-- park.png
|   |-- street_signs.png
|-- microsoft_ocr.py
```

Inside the `config`, we have our `microsoft_cognitive_services.py` file, which stores our subscription key and endpoint URL (i.e., the URL of the API we're submitting our images to).

The `microsoft_ocr.py` script will take our subscription key and endpoint URL, connect to the API, submit the images in our `images` directory for OCR, and then display our screen results.

17.2.3 Creating Our Configuration File

Ensure you have followed Section 17.2.1 to obtain your subscription keys to the MCS API. From there, open the `microsoft_cognitive_services.py` file and update your `SUBSCRPTION_KEY`:

```
# define our Microsoft Cognitive Services subscription key
SUBSCRIPTION_KEY = "YOUR_SUBSCRIPTION_KEY"

# define the ACS endpoint
ENDPOINT_URL = "YOUR_ENDPOINT_URL"
```

You should replace the string `"YOUR_SUBSCRPTION_KEY"` with your subscription key obtained from Section 17.2.1.

Additionally, ensure you double-check your `ENDPOINT_URL`. At the time of this writing, the endpoint URL points to the most recent version of the MCS API; however, as Microsoft releases new versions of the API, this endpoint URL may change, so it's worth double-checking.

17.2.4 Implementing the Microsoft Cognitive Services OCR Script

Let's now learn how to submit images for text detection and OCR to the MCS API.

Open the `microsoft_ocr.py` script in the project directory structure and insert the following code:

```
1  # import the necessary packages
2  from config import microsoft_cognitive_services as config
3  import requests
4  import argparse
5  import time
6  import sys
7  import cv2
```

Note on **Line 2** that we import our `microsoft_cognitive_services` configuration to supply our subscription key and endpoint URL. We'll use the `requests` Python package to send requests to the API.

Next, let's define `draw_ocr_results`, a helper function used to annotate our output images with the OCR'd text:

```
9  def draw_ocr_results(image, text, pts, color=(0, 255, 0)):
10     # unpack the points list
11     topLeft = pts[0]
12     topRight = pts[1]
13     bottomRight = pts[2]
14     bottomLeft = pts[3]
15
16     # draw the bounding box of the detected text
17     cv2.line(image, topLeft, topRight, color, 2)
18     cv2.line(image, topRight, bottomRight, color, 2)
19     cv2.line(image, bottomRight, bottomLeft, color, 2)
20     cv2.line(image, bottomLeft, topLeft, color, 2)
21
22     # draw the text itself
23     cv2.putText(image, text, (topLeft[0], topLeft[1] - 10),
24         cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)
25
26     # return the output image
27     return image
```

Our `draw_ocr_results` function has four parameters:

i. `image`: The input image that we are going to draw on.

ii. `text`: The OCR'd text.

iii. `pts`: The *top-left*, *top-right*, *bottom-right*, and *bottom-left (x, y)*-coordinates of the text ROI

iv. `color`: The BGR color we're using to draw on the `image`

Lines 11–14 unpack our bounding box coordinates. From there, **Lines 17–20** draw the bounding box surrounding the text in the image. We then draw the OCR'd text itself on **Lines 23–24**.

We wrap up this function by returning the output `image` to the calling function.

We can now parse our command line arguments:

```
29 # construct the argument parser and parse the arguments
30 ap = argparse.ArgumentParser()
```

```
31 ap.add_argument("-i", "--image", required=True,
32     help="path to input image that we'll submit to Microsoft OCR")
33 args = vars(ap.parse_args())
34
35 # load the input image from disk, both in a byte array and OpenCV
36 # format
37 imageData = open(args["image"], "rb").read()
38 image = cv2.imread(args["image"])
```

We only need a single argument here, `--image`, which is the path to the input image on disk. We read this image from disk, both as a binary byte array (so we can submit it to the MCS API, and then again in OpenCV/NumPy format (so we can draw on/annotate it).

Let's now construct a request to the MCS API:

```
40 # construct our headers dictionary that will include our Microsoft
41 # Cognitive Services API Key (required in order to submit requests
42 # to the API)
43 headers = {
44     "Ocp-Apim-Subscription-Key": config.SUBSCRIPTION_KEY,
45     "Content-Type": "application/octet-stream",
46 }
47
48 # make the request to the Azure Cognitive Services API and wait for
49 # a response
50 print("[INFO] making request to Microsoft Cognitive Services API...")
51 response = requests.post(config.ENDPOINT_URL, headers=headers,
52     data=imageData)
53 response.raise_for_status()
54
55 # initialize whether or not the API request was a success
56 success = False
```

Lines 43–46 define our `headers` dictionary. Note that we are supplying our `SUBSCRIPTION_KEY` here — now is a good time to go back to `microsoft_cognitive_services.py` and ensure you have correctly inserted your subscription key (otherwise, the request to the MCS API will fail).

We then submit the image for OCR to the MCS API on **Lines 51–53.** We also initialize a Boolean, `success`, to indicate if submitting the request was successful or not.

We now have to wait and poll for results from the MCS API:

```
58 # continue to poll the Cognitive Services API for a response until
59 # either (1) we receive a "success" response or (2) the request fails
60 while True:
61     # check for a response and convert it to a JSON object
```

```
62      responseFinal = requests.get(
63          response.headers["Operation-Location"],
64          headers=headers)
65      result = responseFinal.json()
66
67      # if the results are available, stop the polling operation
68      if "analyzeResult" in result.keys():
69          success = True
70          break
71
72      # check to see if the request failed
73      if "status" in result.keys() and result["status"] == "failed":
74          break
75
76      # sleep for a bit before we make another request to the API
77      time.sleep(1.0)
78
79  # if the request failed, show an error message and exit
80  if not success:
81      print("[INFO] Microsoft Cognitive Services API request failed")
82      print("[INFO] Attempting to gracefully exit")
83      sys.exit(0)
```

I'll be honest — polling for results is not my favorite way to work with an API. It requires more code, it's a bit more tedious, and it can be potentially error-prone if the programmer isn't careful to `break` out of the loop properly.

Of course, there are pros to this approach, including maintaining a connection, submitting larger chunks of data, and having results returned in *batches* rather than *all at once.*

Regardless, this is how Microsoft has implemented its API, so we must play by their rules.

Line 60 starts a `while` loop that continuously checks for responses from the MCS API (**Lines 62–65**).

If we find the text `"analyzeResult"` in the `result` dictionary's keys, we can safely `break` from the loop and process our results (**Lines 68–70**).

Otherwise, if we find the string `"status"` in the `result` dictionary *and* the status is `"failed"`, then the API request failed (and we should `break` from the loop).

If neither of these conditions is met, we `sleep` for a small amount of time and then poll again.

Lines 80–83 handle the case in which we could not obtain a result from the MCS API. If that happens, then we have no OCR results to show (since the image could not be processed), and then we exit gracefully from our script.

Provided our OCR request succeeded, let's now process the results:

```
85  # grab all OCR'd lines returned by Microsoft's OCR API
86  lines = result["analyzeResult"]["readResults"][0]["lines"]
87
88  # make a copy of the input image for final output
89  final = image.copy()
90
91  # loop over the lines
92  for line in lines:
93      # extract the OCR'd line from Microsoft's API and unpack the
94      # bounding box coordinates
95      text = line["text"]
96      box = line["boundingBox"]
97      (tlX, tlY, trX, trY, brX, brY, blX, blY) = box
98      pts = ((tlX, tlY), (trX, trY), (brX, brY), (blX, blY))
99
100     # draw the output OCR line-by-line
101     output = image.copy()
102     output = draw_ocr_results(output, text, pts)
103     final = draw_ocr_results(final, text, pts)
104
105     # show the output OCR'd line
106     print(text)
107     cv2.imshow("Output", output)
108     cv2.waitKey(0)
109
110 # show the final output image
111 cv2.imshow("Final Output", final)
112 cv2.waitKey(0)
```

Line 86 grabs all OCR'd lines returned by the MCS API. **Line 89** initializes our `final` output image with all text drawn on it.

We start looping through all `lines` of OCR'd text on **Line 92.** We extract the OCR'd `text` and bounding box coordinates for each `line`, followed by constructing a list of the *top-left*, *top-right*, *bottom-right*, and *bottom-left* corners, respectively (**Lines 95–98**).

We then draw the OCR'd text line-by-line on the `output` and `final` image (**Lines 101–103**). We display the current line of text on our screen and terminal (**Lines 106–108**) — the final output image, with all OCR'd text drawn on it, is displayed on **Lines 111–112.**

17.2.5 Microsoft Cognitive Services OCR Results

Let's now put the MCS OCR API to work for us. Open a terminal and execute the following command:

```
$ python microsoft_ocr.py --image images/aircraft.png
[INFO] making request to Microsoft Cognitive Services API...
```

```
WARNING!
LOW FLYING AND DEPARTING AIRCRAFT
BLAST CAN CAUSE PHYSICAL INJURY
```

Figure 17.1 shows the output of applying the MCS OCR API to our aircraft warning sign. If you recall, this is the same image we used in Chapter 16 when applying the Amazon Rekognition API [58]. I included the same image here in this chapter to demonstrate that the MCS OCR API can correctly OCR this image.

Figure 17.1. Image of a plane flying over a warning sign. OCR results displayed on the sign in green.

Let's try a different image, this one containing several challenging pieces of text:

```
$ python microsoft_ocr.py --image images/challenging.png
[INFO] making request to Microsoft Cognitive Services API...

LITTER
EMERGENCY
First
```

```
Eastern National
Bus Times
STOP
```

Figure 17.2 shows the results of applying the MCS OCR API to our input image — and as we can see, MCS does a *great* job OCR'ing the image.

Figure 17.2. *Left:* Sample text from the First Eastern National bus timetable. The sample text is a tough image to OCR due to the low image quality and glossy print. Still, Microsoft's OCR API can correctly OCR it! *Middle:* The Microsoft OCR API can correctly OCR the *"Emergency Stop"* text. *Right:* A trashcan with the text "*Litter*." We're able to OCR the text, but the text at the bottom of the trashcan is unreadable, even to the human eye.

On the *left*, we have a sample image from the First Eastern National bus timetable (i.e., schedule of when a bus will arrive). The document is printed with a glossy finish (likely to prevent water damage). Still, due to the gloss, there is a significant reflection in the image, particularly in the *"Bus Times"* text. Still, the MCS OCR API can correctly OCR the image.

In the *middle*, the *"Emergency Stop"* text is highly pixelated and low-quality, but that doesn't phase the MCS OCR API! It's able to correctly OCR the image.

Finally, the *right* shows a trashcan with the text *"Litter."* The text is tiny, and due to the low-quality image, it is challenging to read without squinting a bit. That said, the MCS OCR API can still OCR the text (although the text at the bottom of the trashcan is illegible — neither human nor API could read that text).

The next sample image contains a national park sign shown in Figure 17.3:

```
$ python microsoft_ocr.py --image images/park.png
[INFO] making request to Microsoft Cognitive Services API...
```

```
PLEASE TAKE
NOTHING BUT
PICTURES
LEAVE NOTHING
BUT FOOT PRINTS
```

Figure 17.3. OCR'ing a park sign using the Microsoft Cognitive Services OCR API. Notice that API can give us *rotated* text bounding boxes along with the OCR'd text itself.

The MCS OCR API can OCR each sign, line-by-line (Figure 17.3). Note that we're also able to compute rotated text bounding box/polygons for each line.

The final example we have contains traffic signs:

```
$ python microsoft_ocr.py --image images/street_signs.png
[INFO] making request to Microsoft Cognitive Services API...

Old Town Rd
```

```
STOP
ALL WAY
```

Figure 17.4 shows that we can correctly OCR each piece of text on both the stop sign and street name sign.

Figure 17.4. The Microsoft Cognitive Services OCR API can detect the text on traffic signs.

17.3 Summary

In this chapter, you learned about Microsoft Cognitive Services (MCS) OCR API. Despite being slightly harder to implement and use than the Amazon Rekognition API, the Microsoft Cognitive Services OCR API demonstrated that it's quite robust and able to OCR text in many situations, *including* low-quality images.

When working with low-quality images, the MCS API *shined*. Typically, I would recommend that you programmatically detect and discard low-quality images (as we did in Chapter 13). However, if you find yourself in a situation where you *have* to work with low-quality images, it may be worth your while to use the Microsoft Azure Cognitive Services OCR API.