CHAPTER 7

Codicology: Automated Analysis Using Python and OpenCV

> Grau, Teurer Freund, ist alle Theorie.
> Und grün des Lebens goldner Baum
>
> GOETHE, *Faust*, 1808. Der Tragödie erster Teil. Studierzimmer, Mephistopheles zum Schüler

Of all aspects of manuscripts studies, codicology has been singled out as relying the most on the actual artifact. Interest in codicology, it is said, will maintain its relevance in having access to the actual manuscript, not just a digital surrogate. So far, advances in 'digital codicology' have confirmed this. I am thinking here of efforts to capture specialized photos that reveal codicological aspects. There are numerous examples of multispectral imaging,[1] hyperspectral imagining,[2] thermographic imagining,[3] and x-ray imaging.[4] Other advanced techniques have also been used, such as scraping off or otherwise

1 E.g. Arsene, C.T.C., P.E. Pormann, N. Afif, S. Church, and M. Dickinson. "High Performance Software in Multidimensional Reduction Methods for Image Processing with Application to Ancient Manuscripts." pp. 1–25 in *Manuscript Cultures*, 2016; Hollaus, F., M. Gau, R. Sablatnig, W.A. Christens-Barry, and H. Miklas. "Readability Enhancement and Palimpsest Decipherment of Historical Manuscripts." pp. 31–46 in *Kodikologie Und Paläographie Im Digitalen Zeitalter* 3. Norderstedt: Books on Demand, 2015; Easton Jr., R.L., and W. Noël. "The Multispectral Imaging of the Archimedes Palimpsest." pp. 39–49 in *Gazette Du Livre Médiéval* 45 (2004).

2 Shiel, P., M. Rehbein, and J. Keating. "The Ghost in the Manuscript: Hyperspectral Text Recovery and Segmentation." pp. 159–174 in *Kodikologie Und Paläographie Im Digitalen Zeitalter*. Norderstedt: Books on Demand, 2009.

3 Meinlschmidt, P., C. Kämmerer, and V. Märgner. "Thermographie—Ein Neuartiges Verfahren Zur Exakten Abnahme, Identifizierung Und Digitalen Archivierung von Wasserzeichen in Mittelalterlichen Und Frühneuzeitlichen Papierhandschriften, -Zeichnungen Und -Drucken." pp. 209–226 in *Kodikologie Und Paläographie Im Digitalen Zeitalter 2*. Norderstedt: Books on Demand, 2010.

4 Deckers, D., and C. Glaser. "Zum Einsatz von Synchrotronstrahlung Bei Der Wiedergewinnung Gelöschter Texte in Palimpsesten Mittels Röntgenfluoreszenz." pp. 181–90. in *Kodikologie Und Paläographie Im Digitalen Zeitalter 2*. Norderstedt: Books on Demand, 2010.

analyzing little pieces for DNA[5] or comparing ancient paint recipes with the paint analysis of surviving illustrations.[6] All these methods rely on the physical manuscript and do not create a digital surrogate that can be used beyond the single analysis it was meant for.

Perhaps conventual codicology can, after all, benefit from using digitized manuscripts, without needing expensive and advanced technology. In this chapter, we will manipulate digitized manuscripts using the programming language Python and the software library OpenCV (both to be explained below), which are free to use and relatively easy to learn. No team is needed, no grant money is required. Just you, your computer, and a bunch of digitized manuscripts. The plan of the chapter is to walk through an actual project from start to finish. If you actively participate by replicating all of the code and by playing around with it, you will be able to run this code yourself. More importantly, by the end of the chapter, you will be able to customize the code and implement your skills in other tasks both within and beyond codicology. In fact, the most important information in this chapter is not specifically how you can do the case study that I present but to learn of the general principles of programming and the basic strategies to figure out a programmable solution. The explanation of Python and OpenCV is only a case study to learn how programming in general works.

As a case study and proof of concept, we will investigate a unique feature of manuscripts produced in the Islamic world, namely their closing flap (*lisān*). Codices from Europe have a front cover, spine, and back cover. However, Islamic manuscripts very often have an additional flap attached to the back cover that folds over the long edge opposite the spine onto the front, thereby giving the codex more protection and integrity. Some notebooks nowadays have this too, falling on top of the front cover to keep it closed. But classical Islamic manuscripts were designed to tuck the flap underneath the front cover. This flap is called a 'tongue' in Arabic, perhaps because of its shape, which is triangular and ends in a tip. With a dataset of several thousands of manuscripts, I set the challenge to automatically detect the angle this triangle makes. This may seem like an uninteresting aspect but that's exactly the point: who knows what kind of information is contained in such an innocent-looking aspect. Perhaps this

5 Stinson, T. "Counting Sheep: Potential Applications of DNA Analysis to the Study of Medieval Parchment Production." pp. 191–207 in *Kodikologie Und Paläographie Im Digitalen Zeitalter 2*. Norderstedt: Books on Demand, 2010; Teasdale, M.D., S. Fiddyment, J. Vnoucek, V. Mattiangeli, C. Speller, A. Binois, M. Carver, et al. "The York Gospels: A 1000-Year Biological Palimpsest." pp. 1–11 in *Royal Society Open Science* 4 (2017).

6 Barkeshli, M. "Material Technology and Science in Manuscripts of Persian Mystical Literature." pp. 187–214 in *Manuscript Cultures* 8 (2015).

angle is always exactly the same, showing some kind of industry-wide agreed upon standard. Maybe it is highly irregular, showing personalized craftsmanship. Such will tell us something about book production in the Islamic world. Maybe there are a couple of standards which can be related to different eras or parts of the world. Such would be valuable information to take into account for manuscripts without a date or place; you measure up the angle of the flap and you have one additional argument to help make an educated guess about the origin.

1 Why Code?

Coding means giving one or more instructions to a computer. Before turning to any programming language as a solution, we need to know if a computer-supported answer is the right solution. There are generally two good reasons. First, there are some things that a computer can figure out which are beyond the capacity of a human being. This can be seen in the case of specialized photography, revealing aspects of a codex that elude the naked eye. Second, there are some things that a computer can figure out for which a human being may not have the patience or stamina. Think of a chess computer that quickly goes through all possible moves and all the moves that each move would possibly result in, and so forth, almost one to two dozen moves ahead: a human being *could* do it, provided they have enough scrap paper and plenty of time. But a computer can do this kind of monotonous work in seconds (grandmasters are typically able to think about five moves ahead). The difference is so huge that the time it takes to write a chess computer that calculates two dozen moves ahead is lesser than to actually calculate two dozen moves ahead yourself. And once it is written, it can be used over and over again. A popular introduction to the programming language Python is called *Automate the Boring Stuff with Python* and this encapsulates quite well the main reason for letting a computer do the heavy lifting for you: it would be too boring to do it yourself. The case study for this chapter falls squarely into this second category. Sure, we could use a *geodreieck* to measure the angle of a flap by hand, write it down, do it again for the next thousand manuscripts, and arrange the resulting angles in a meaningful way, for example, by counting them all up and dividing by thousand to get the average. But it would take an unjustifiable amount of time and its results could be repurposed only in a limited way. Furthermore, quality assurance relies solely on judging the reliability of the person measuring the angle. With a computer running a program, the answer can be obtained much faster, both the program and the results can be used and repurposed in future

research, and the quality can be checked by judging the code and running tests and examples with known solutions.

Once you are convinced that writing a computer program is a good technique to get the answer to your research question, you will likely need to adjust your question (and your solution) to make it programmable. At the time that I am writing this, we live in the age of *smart devices*; pieces of ordinary technology beefed up with features that are designed to assist you based on your specific situation. Apple's 'Suggestions' feature and Google's 'Autocomplete' feature literally attempt to guess how you want to finish your sentence. At the core, a computer is anything but smart. It is very dumb indeed, and it will do only and exactly what you tell it to do. Furthermore, it operates on a binary principle: everything must be either yes or no, on or off, one or zero. 'All theory is black or white,' says Goethe, and so rings the paradigm of the age of computers. However, life around us presents itself in all its vivid colors. In other words, we need to convert that colorful real-life problem into a black-and-white digital problem. In our case study, we do so literally. We literally transform the color image of a manuscript cover into a black and white object living only in the virtual memory of a computer, which is deleted the moment the computer finishes running the code. For humanities research, this approach can be frustrating. Instead of saying 'it is' or 'it isn't,' we often times say 'it is kind of,' 'it is similar to,' or 'it could be.' Both our research input and output present the quality of life in all its vivid colors. Take a photo of a manuscript cover, for instance. Classically trained as we are, we instantly recognize the shape of the codex, the surface on which it is resting, the colors, materials, and embellishments. We are so good at this that even if it is severely damaged, we can almost instantaneously make out all these features. We see them as they are and can easily make subjective judgments of the quality, beauty, and even the authenticity of the codex. A computer, on the other hand, sees nothing of this. All the computer 'knows' is the color code for every pixel of which the image is made up. A computer cannot 'see' the manuscript through the pixels, the proverbial forest through the trees.

To figure out if the real-life, colorful question can be turned into a digital, black-and-white question, it is a good practice to collect all information available about our starting position and about the desired end result. A good rule of thumb is to try to transform every bit of qualitative information into quantitative information. Once the start and end positions are clear, you can write down the intermediate steps to figure out how one could get from the start to the end. This does not have to be very detailed, as you will repeat this very same process multiple times on more detailed levels to figure out the actual steps needed between the two steps of the general path. This activity is called

'writing pseudo-code.' Pseudo-code is a rather colorful step-by-step description of what ought to be done, using vocabulary, grammar, and punctuation of a human language such as English. It is the blueprint for writing the actual code, the black-and-white description written in the vocabulary, grammar, and the punctuation of a computer language (such as Python).

2 Description of Case Study

This case study began as a brainstorming session on what to do with access to a couple of thousand digitized manuscripts, all stored as PDFs in which each page contained one photo of a page-spread of the manuscript. Several ideas took root. Chiefly, they were text block analysis, seals analysis, and codex analysis. Text block analysis could consist of measuring its size or its number of lines, which would yield interesting metadata of its own and would be a first step towards automated analysis of the text contained in the manuscripts. Seal analysis would focus on detection of same or similar seals across a corpus. Since seals are mechanically constructed they are supposed to look exactly the same at all times and that seems like an easily exploitable feature for automated searching. Its results will be significant for ascertaining the provenance and ownership of a manuscript. The digital research on them has so far been limited to handmade databases.[7] Codex analysis could focus on medallion-shaped mandorla ornamentation (*lawza, shamsa* or *jāma* in Arabic) or the flap (*lisān*).

A subset of 2500 manuscripts, of the Nuruosmaniye collection, included a photo of the cover as a first page of the PDF. It was decided to focus on flap, specifically its angle. It is possible to extract the angle from a photo without knowing the true dimensions of the codex, since an angle stays the same no matter how you scale the two lines forming it, hence the number of variables is relatively low, making this case study relatively straightforward. Also, it may be noted that this aspect has never been rigorously studied, thereby providing an opportunity to explore a research question never before asked.

At this early stage, a two-step plan emerged: (1) extract the correct image from a PDF file, and (2) analyze the image to find the angle. A logical third step would be to store and process the resulting angle. However, since this was considered to be of lesser difficulty, it was at first left out of the plan.

For step one, I searched for a package (or library) that would handle PDFs within the ecosystem of the programming language Python. A package is a

7 The example I was thinking of, Chester Beatty's Islamic Seals Database, has been taken offline sometime during 2018. The digital world has precarious life.

pre-written block of code that you can download and use for free. With 'ecosystem' we mean all available packages combined, especially as they are indexed or discussed on major online resources for the particular programming language. For us budding programmers, it is terribly important to leave the heavy lifting to the professionals. PDFs are a prime example of that; once you search around for automatic PDF manipulation, you learn quickly that as nice and shiny PDFs look in a viewer, they can be an absolute mess under the hood. Broken down into its bits and bytes, understanding PDFs is only doable after investing a lot of time, which is why it is important to use a package. There were a few choices, all of them not very well documented, and I landed on PyPDF2, more on which later.

For step two, I then searched for an image manipulation package. I reviewed some candidates, trying to figure out which package was popular, still actively developed, well-documented, and would actually do what I needed. I was convinced by OpenCV, especially after seeing demonstrations of it, for example its use in programming a self-driving car for the popular video game *Grand Theft Auto*. It seemed to have a low barrier to entry while also packing advanced options, assuring me that I would be able to learn it and then keep using it as my use became more demanding.

3 Introduction to Python

The choice for Python for use in digital humanities is easy. From a technical point of view, what Python has got going for itself is that it is a high-level computer language. This means that it looks a lot like actual English, which makes it easy to read and write. It also means that it takes care of a lot of things without requiring the user's explicit command. A notable example is the automatic garbage collection. Your code will likely need to temporarily store virtual files. For example, when we direct our computer to open a PDF file, this PDF is opened in the computer's memory, its RAM. When things are no longer used, Python automatically erases them from the computer's memory, freeing up memory for future tasks. Other programming languages would not do anything until you, the programmer, told them to remove all unused items from memory. Moreover, Python works on pretty much any platform (Windows, macOS, Linux, Android, and so on) which makes it easy to carry over your work, or to integrate somebody else's work.[8] Other programming languages are

8 The reality is more complicated but also exponentially more technical which is why I do not go into it. This chapter is meant as a primer in Python, not a specific and rigorous introduction.

very close to Python in these regards, a notable example being Ruby. However, a decisive argument in favor of Python is its ecosystem. Every programming language has its own ecosystem of prewritten libraries and packages that you can freely make use of. But because this collection is created by its users, it can have some use cases that are very well covered and others quite poorly. Python has an excellent collection of packages for many of the tasks we would want to perform in the humanities. This is true for our purpose of image manipulation, but perhaps even more so once we have the texts we want to study in digital format and want to perform text analysis on it.

From a practical point of view, we may note the maturity and popularity of Python in general and among scholars of the humanities in particular. This means that Python is a reliable language that will not fall into obscurity anytime soon and render our skills in it useless. It also means that the language itself works wonderfully, without bugs. More importantly, it means that documentation, training, tutorials, and questions and answers are plentiful, in the form of websites, books, and videos. If you do not know how to do something, searching for it online will likely find you with the details of a conversation between somebody who wanted to do (almost) the same and somebody who explains how it can be (or why it cannot be done) on a website like stackoverflow.com. It is worthwhile to become an active member of Python's user community, engaging in current conversations and beginning new ones if your specific question has never been answered. Quite often, you will find that the information you are after is available in different formats, geared to different levels of experience. The popularity further means that there are a lot of packages and example code available on websites like pypi.python.org and github.com. Specific implementations for digital humanities can be found in greater instances than other programming languages, which makes it easier to hit the ground running and share and discuss active projects. Coding means to have a web browser open at all times, on some page of the documentation of Python or OpenCV, on StackOverflow, on some enthusiastic blogger who explains his experiences, or perhaps Wikipedia. Searching, reading, trying, repeat. There is no point in reinventing the wheel.[9]

Let us discuss how to actually *do* Python. Whatever happens, now that you have decided to actually start programming, you should reassure yourself that you have made the right decision. There will be plentiful frustrating moments

9 At the same time, there will be moments where it will be easier to write your own functionality rather than trying to adapt an existing one. In programming jargon, this is called 'rolling your own'. With experience you will be able to judge the solutions that are currently out there and see if they exactly fit your needs and if not, how much time it would take to make the required changes versus the time it could take to write it from scratch.

when your computer is not doing what you want it to do. For every problem there is a solution, and it is not until a much more advanced stage that you will be the first one to encounter a newfound problem. Since many people have gone before you, they have already found the solution to your problem and talked about it on the internet and in books. Learning a programming language with no real prior experience is not going to happen overnight. So it is good to keep telling yourself that you are not 'losing' time over finding a fix for whatever is going wrong, because this is time invested in better understanding what programming is and how computers can and cannot help you.

To get going requires some initial setting up. Since all of this is a one-off type of work, you might as well have an experienced user (in your local DH discussion group or DH Lab at your faculty or library) do it. Moreover, since the process and requirements may change over time, I shall refrain from a detailed description. What does bear mentioning here is that Python requires *installation*. The way you can have your computer do anything through the Python programming language is by installing some applications from Python.org. They come down to two: a standard library of functions (think: a minimum vocabulary) and an interpreter (think: somebody who listens to your commands in the high-level Python language and tells the computer in a low-level computer language what to do). Once installed, you can write code in the Python language in any plain text editor and save the file as FILENAME.py. There is no graphical user interface for Python. Instead, you use a command line to run the Python interpreter and tell it to read your FILENAME.py. Your computer is now executing your commands!

Confusingly, there are two versions of Python in use. In 2008, Python 3 was released which has been steadily developing. In that same year, a new version of Python 2 came out, 2.6. Version two and three are truly different and the code of one version will not run in the other version. To highlight the popularity of Python 2, even after the release of Python 3.1 (in 2009), Python 2.7 came out (in 2010). That was the final release of Python 2 and is, until today, widely used. I use Python 3 myself and recommend it because it is more future-proof. However, be aware that many examples of code you will find on the internet were written for Python 2. This means that you need to be well aware of which version is installed on your computer and, if both, which version is running (interpreting) your script. Last, be absolutely sure that the packages you install using a package manager like *pip* are installed for the specific version of Python that you are using. You can specify the version of Python like so: *python3 -m pip* followed by what you like to do (most likely *install PackageName,* or *list*, to see which packages are installed). In case of any doubt, seek help from a more experienced user to set things up for you and be sure to document it so that you can use it by yourself later.

If you think writing code in a simple text editor and then running it in the terminal seems awkward, you are not alone. There are tools to make life easier. I recommend using a so-called IDE (Integrated Development Environment) which allows you to write Python code, save it, run it, and even troubleshoot it, and all of this in an application that has a lot more graphical elements than a mere command line. *PyCharm Community Edition* is my current choice, which is a free, easy-to-use, and yet advanced application.[10] PyCharm actually knows the vocabulary, grammar, and punctuation of Python and will help you write your code by suggesting which functions to use and which variables need to be specified, by color coding the grammar of your script, and by indicating where you have made a punctuation mistake. It will allow you to run your scripts with a mere click or keystroke and give helpful feedback in case of an error, enabling you to debug your code more easily. For Orientalists: it supports Arabic flawlessly, which cannot be said of all code editors.[11]

4 Introduction to OpenCV

For our project, you will need to install a couple of packages: PyPDF2, NumPy, and OpenCV. OpenCV is an open source package that has functionality for what is called 'computer vision.' When we look at a photo, we can immediately make out the different objects and their features; it would be hard indeed to see a photo as a mere collection of blobs of color. For a computer, this is the other way around; when a photo is loaded into memory, all that is loaded is a color code for each pixel. The photos I use for this project vary in size, but all of them are built up from more than 3.5 million pixels, sometimes more than 5 million pixels. When a photo like that is loaded in memory, all a computer 'sees' is 5 million color numbers. To be more precise, when OpenCV opens an image, what is loaded into the computer's memory is a matrix with each field containing a vector of three dimensions. That is mathematical speak for what can be described as a table where the number of columns and rows correspond to the number of pixels indicating the image's width and height, as though we drew a fine raster over the image, with each field in that table containing a traffic light. Instead of Red, Yellow, and Green as an actual traffic light, these 'traffic lights' have Blue, Green, and Red. And instead of the three lights following each other, they are all in the same spot and their mixture creates one specific

10 In this chapter I have not made use of Jupyter but it is also a highly recommended tool to have a more graphically rich interface to write Python, with easy functionality to test and tinker with your code.

11 For example, the popular code editor *Sublime Text* does not support Arabic correctly.

color. Normally, these blue, green, and red colors are divided into 256 shades (from 0 to 255). As such, sixteen million different colors can be composed from mixing blue, green, and red. In OpenCV, this is called the BGR value. You may have heard of RGB, which is how most software handles color. While OpenCV has it the other way around, there is no point in being upset about this. We simply need to remember this fact when we read out a value of the table that represents the image. For example, if we ask OpenCV the color of a pixel, for example the one that is 100 pixels to the right and 100 pixels down from the top-left corner, we will get an answer that can look like (255,0,0). This means that the pixel is pure blue (and not pure red!).

To manipulate such a matrix by hand would be very tedious and complicated. The power of OpenCV is that there are all kinds of functions that perform sweeping transformations of that matrix. One basic operation that is almost always required before doing other things is to reduce the image to a grey-scale image. Instead of each space in the matrix having a three-color traffic light, the spaces now only have one light, which is defined along 256 shades of grey, 0 being pure black and 255 being pure white. This greatly reduces the complexity of an image and makes available a large number of analyses.

We codicologists are of course not the first to want to automate the analysis of photos. Video surveillance in gas stations, restaurants, streets, and almost anywhere, has given rise to a demand for automatic analysis, for example, to give a live count of how many people are in the store. The automatic processing of forms such as cheques and surveys is another example. OpenCV is the result of many years of development towards that end. Us humanists can piggyback on the advances of this industry. However, you will notice quickly that even though OpenCV has many powerful techniques that require only one or two lines of code, it requires a bit more skill to operate than Python itself. There are only a few tutorials online, many of them are written for the implementation of OpenCV in another programming language (C++), and even the official documentation is sparse and sometimes incomplete. Nonetheless, the power of OpenCV is worth the extra effort.

Instead of opening a digital photo with a viewer to make the photo appear on our screen, we now tell OpenCV to open it and are then able to either perform some sort of detection or transformation. As computers are great at answering yes or no questions, OpenCV is great at answering a black or white question. There is no standard function in OpenCV to detect the angle of a manuscript. But if we can reduce those 5 million pixels to the three pixels that define the corners of the flap, then we can get the angle with some simple mathematics. Again, there is no direct, magic function to get to those three pixels. Instead,

we need to gradually reduce the number, and do so in a fashion that takes into account all the irregularities (the vivid colorfulness) of the manuscripts. When we have extracted the image from the PDF, we will therefore first apply a number of transformations aimed at reducing the color image into an image in which the background is black and the manuscript is white, paying particular attention to the flap being as solidly white as possible.

5 Step 1: Extraction of Images

In Python scripts, you can include comments that the computer will skip by beginning the line with a *hash* #. It is a good custom to begin code with a signature and a short remark about the use of the script. This script is meant to do the following: it opens a PDF document, takes a page, and saves the image contained on that page as a JPG image.

A computer reads a script line by line: it knows about the lines it has read but has no idea about the lines ahead. Therefore, if you require packages at some point, it is best practice to have the computer load them in at the very beginning so that you are assured that the computer knows about it and, consequently, can use it. This also makes the script more ordered for yourself. In this case, we will need one function from the IO package which gives additional features for *input/output*, i.e., for handling interactions with files on a hard drive. PyPDF2 is the package we downloaded specifically to interact with PDF files. With the function *PdfFileMerger*, we can create virtual PDF files. Since we do not need more, we only need to load those specific functions.

After the packages, I like to declare any variables that are actually used as constants.[12] A variable is a word that can stand in for some value. Programming languages maintain different types of values, such as a string, integer, or boolean. Python does too; but you do not have to declare the type when you define a variable, it will simply adapt to your use. A wrong type can still cause an error. The simplest example is the *print* command which outputs any selected text. This command cannot take a combination of strings (letters) and integers (numbers). So, if you need to mix them, you first need to convert the integer to string with the *string*() command. Variables are very useful because you can keep updating its value and create a different result because of that new value.

12 Unlike other programming languages, there is currently no true support for constants in Python. The way to define a constant is to simply define a variable and make sure yourself to never change it.

```
# L.W. Cornelis van Lit, O.P. (c) 2018
# Function for fetching a specific page from PDF and saving
as JPG
# ----------------------------------------------------------------

# Dependencies
from io import BytesIO
from PyPDF2 import PdfFileMerger

# A string defining a JPG always starts and ends with these
values
startMark = b"\xff\xd8"
endMark = b"\xff\xd9"

# Predefined path where files are
pathOfFiles = "/Volumes/ExternalHardDrive/FolderName/SubFolderName/"
nameOfCollection = "NURUOSMANIYE"
startNumber = 1000
finishNumber = 2000

# ----------------------------------------------------------------
# Check if a file exists and is accessible.
def Accessing_file(filePath):
  try:
    file = open(filePath, "rb")
    file.close()
    return True
  except:
    return False

# ----------------------------------------------------------------

# This function takes one PDF file and extracts one page from
it, to save as JPG.
# Only to be used on PDFs in which each page is only an image.

# Check if file exists. If not, go to next.
def Save_page_PDF(nameOfPDF, pageOfPDF):
  if not Accessing_file(pathOfFiles + nameOfPDF + ".pdf"):
    return
```

We will see examples of this later. Why, then, should we declare variables that never change their value? I do this for readability, and this particular case is a good example. Normally, you would open a PDF document with a program that would display the document on your screen. But this time, we want the computer to open it. This means that it will read the PDF file simply as a long series of characters (a PDF viewer is only translating those characters into differently colored pixels on your screen). Our strategy for obtaining the image from the specific page of the PDF file is to look for a pattern in that series of characters that would indicate that this particular section of the series defines an image. By opening one PDF in a PDF viewer and investigating the elements of a page, I came to the conclusion that the images that made up the PDF were likely JPG images. Searching the internet reveals that JPG images, when read as a long series of characters, always start with *b"\xff\xd8"* and end with *b"\xff\xd9"*. If we can get the specific page of a PDF as a series of characters, we can look for those parts to identify the image. More on the specifics of that strategy later, for now, notice that using *b"\xff\xd9"* in the actual code looks awkward and may be confusing when you revisit the code after many months. Moreover, if you share it with somebody else, it will likely perplex the other person to leave in the seemingly random string of characters *b"\xff\xd8"*. Besides, if we use it more than once, we will be prone to make a typing mistake. Instead, in this code, we can substitute it for a word that makes sense to anybody who speaks English, for instance variable names like *startMark* and *endMark*. All my variables start with a small letter and then each next word is written immediately attached to it but with the first letter capitalized. Such a convention is solely mine and not part of Python's grammar, but it does help for readability.[13]

After the constants, we declare variables that are variable but only in as much as we will change them by hand, while the code never changes them but only uses them. For this script, the only such variable that needs to be declared is the one defining the path to the PDF file. You may have noticed that we split the path into different components, which will come in handy as we reuse some parts to make it easier to iterate the extraction code over many PDF files and also to construct a meaningful file name for the extracted JPG image. Quotation marks indicate that a variable has a string value; and to use all the variables to put together a path to the PDF file, we need all variables to be a string. You might notice that *startNumber* and *finishNumber* do not have quotation marks and that means they are integer values. We will use them as integers as well as strings and will only convert them to string when needed.

13 There are style guides that can help you with this. For a deep-dive into writing conventions a good start would be 'PEP 8', the style guide for Python.

```
# File exists, so prepare the virtual containers.
  merger = PdfFileMerger()
  virtualpdf = BytesIO()

# Opens PDF and takes out specific page
  with open( pathOfFiles + nameOfPDF + ".pdf", "rb") as
sourcePDF:
    merger.append(fileobj=sourcePDF, pages=(pageOfPDF - 1,
pageOfPDF))
    merger.write(virtualpdf)
  merger.close()

# Read the desired page simply in its string of values
  pdf = virtualpdf.getvalue()
  virtualpdf.close()

# Find the start and end of the string defining the JPG
  jpgStart = pdf.find(startMark, 0)
  jpgEnd = pdf.find(endMark, jpgStart)

# Reading out the entire string defining the JPG
  jpgString = pdf[jpgStart:jpgEnd]

# Save the JPG to the hard drive
  with open(nameOfPDF + ".jpg", "wb") as jpgFile:
    jpgFile.write(jpgString)

# ------------------------------------------------------------
----

# Loop the function
for i in range(startNumber,finishNumber+1):
  print("Working on Manuscript" + str(i))
  SavePagePDF(nameOfCollection+str(i),1)
```

pathOfFiles can be left empty or, rather, defined as only two quotation marks.[14] This would make the script look for the PDF file relatively to which folder the Python script is in. In the code shown here, an absolute path to the PDF file is given. The example given here works for macOS. For Windows, the absolute path would start with the drive letter instead of */Volumes*. With some effort, it would be possible to construct variables and code that would be system independent, where all the user has to do is fill in the drive name and folder path and the code would work no matter which platform you are on. However, there is no reason to do that here. In general, you write code until a level of functionality that works for you and no further. Going further would be a waste of our time and likely also of processing power. Since we will eventually run this script over hundreds, possibly thousands, of PDF files, straining the computer becomes a real issue. We are nowhere near a skill level that we can actually and intently optimize our scripts to run the fastest. However, being aware that with every line of code we add processing time already helps. More important, however, is the first point that it would be a waste of our own time to figure out how to add functionality that we in practice will not use. The flip side to this is that we might want to use the script in the future for a different purpose or even share it with others. Two principles can support this; first, to include comments in your code and, second, to set things up flexibly using variables and functions that can easily be altered or swapped in/out. My approach is, then, close to the programming paradigm called *procedural programming*. Other modern paradigms are *object-oriented programming* and *functional programming*. I use the other paradigms as well in an interspersed manner, as we will see. It is a matter of complexity and usability that will drive your exact decision how to code. You will undoubtedly choose the path of least resistance. But do keep in mind that what seems like a quick and dirty fix right now may prove to be a major rewrite later on when you want to reuse or share your code. Indeed, reopening your code after a long hiatus will make it look like someone else's code and you will thank yourself for having used clear and useful variable names and having documented the functionality along the way.

We now come to our actual code, which consists of two functions, one called *Accessing_file* and the other *Save_page_PDF*. For function names, I use the convention of starting with a verb which is capitalized and each subsequent word is separated by an underscore and is written in lowercase. Since

14 As is often the case, programming allows you multiple paths towards the same goal. For directing Python to the right file you could also try to store the path in a so-called f-string.

'PDF' is an acronym and normally written in capitals, I have done so too for the name of the function.

What are functions? They are basically scripts within a script; several lines of code that are not activated by telling the computer to run the script but which have a name and can be called upon as many times as you like within the script. You can put something in a function and, normally, something (else) comes out. It is quite normal to first simply code line after line to get what you want and at the end tidy everything up by placing it in a function and cleaning up all the variable names. Splitting things into different functions is helpful in distinguishing different tasks. For example, the checking of the soundness of a file is a task different from the extraction of an image from a file. An additional benefit is that within PyCharm CE, you can quickly close/open a function with the +/- sign in the margin to retain an overview of your code. If your code becomes even larger and more complex, you can also split your code over different files and load them in as modules. In Python, you can define a function by writing *def NameOfFunction(Variables):*. Take *Accessing_file* as an example. Its function is merely to check if a file exists and is not corrupt. A nonexistent or corrupt PDF file, after all, is useless to us. If we were to try to extract a page from such a PDF, the code would give an error. In that case, what we want to put into the function *Accessing_file* is the path to the file. I wrote the script such that we can use the function both for relative and absolute paths, a decision that we can make later. We can call and activate the function anywhere in our script by writing *Accessing_file*(*PathToAFile*) with *PathToAFile* being an actual path to an actual file, for example *Nuruosmaniye/Nuruosmaniye1.pdf*. In the place where the function is defined, we notice *filePath* placed within brackets; this is a variable, and by calling the function and giving an actual path to a file within brackets, we fill that variable with a value. We can, then, within that function use that variable.

Within *Accessing_file*, I use a bit of object oriented programming. I create an object called *file* and use the *open* function. This itself takes two parameters, a path and a mode. By path is meant the actual path to the desired file, which within our function is represented by the variable *filePath*, so that is what we write. The mode is a more technical aspect: do we want the computer to only have reading or writing access, or both? We only need reading access, so we use *r*. The extra *b* is an even more technical detail that tells the computer to read the file as a binary file, not a text file. Since PDF and JPG files are indeed not text files (more on that in Chapter Five), we want to ensure that the computer interprets those files correctly. Once the object *file* has been created, we can close it again, since all we wanted to know is if the file existed and is able to be

opened. It is good practice to explicitly use a *close* command to make sure we are not leaving unnecessary things in memory. Last, if indeed the file is openable, we want to return a value of *True*. If we call the function and give it a path to an existing file that can be opened, we will get an answer of *True*. Had we not wrapped the foregoing lines of code in a *try/except* construction, the code would still give an error if the file or path would not be correct. Therefore, we tell the computer to only *try* to open the file. If that works, it will go on to the next line and close the file, and then move to the next line where it will return to the point where the function was called and return *True*. If, however, it cannot open the file, it will go to the *except* part of the code since an exception has been encountered. There, it will see that it needs to return to the point where the function was called and return the value of *False*. True and false, here, act as a yes or no to the question: is the file accessible? You might still be unsure about how to use this function or if it is even required. We will come to that when we discuss the next function.

The core part of our script is the function *Save_page_PDF*, which needs the name of a PDF file and the page number which we want to extract. In this chapter, we wish to extract the page with the photo of the cover of the manuscript, which is almost always on page one. So we will mostly call this function with 1. But the extra flexibility was easily built and may prove useful later when we have a sudden need to extract all page twos or threes. *Save_page_PDF* begins with an *if*-statement. This is probably the most foundational building block of programming. With an *if*-statement, you can perform certain code if a certain condition is met. You can even add *else-if*-statements (written in Python as *elif*) or a catch-all *else*-statement. In Python, the grammar and punctuation of an *if*-statement is much like defining a function: you begin with a trigger word (for an *if*-statement it is 'if', for a function it is 'def'), then the line must end in a colon. All the lines that should be executed within that *if*-statement (or function) should be indented by one tab relative to the *if*-statement. Thus, if you have functions or *if*-statements nested inside each other, you add additional tabs counting from the start of the line. You do not need to declare the end of a function or *if*-statement; Python knows it from where the code is not indented any longer since it assumes that lines that have the same indentation are in the same block of code. Between *if* and the colon goes the condition that is to be evaluated. A simple condition would be 'if VARIABLEX contains a higher value than a CERTAINNUMBER …'. An *if*-statement is ideal for a yes/no question. In fact, if you simply put a variable in between *if* and :, Python will check if that variable is *True*. You can also invert the same by writing *not* before it. In our code, we put the function *Accessing_file* in between *if* and :. We will

thus check if *Accessing_file* returns *True*. In this case, we use the *if*-statement to catch files that are nonexistent or inaccessible. If so, we want to immediately abort and exit the function. So instead of looking for cases where *Accessing_file* gives *True*, we want to look for cases where it gives *False*. This is why we write *if not Accessing_file* … Notice the construction of the path to the file from three components: a static variable, a dynamic variable, and a string.

It is entirely possible to do this differently. For example, right now, *Accessing_file* gives *True* if a file exists and is accessible. But we could just as well make it return *False* and then the *if*-statement would not have needed the *not*. Additionally, there are many other ways to do it too. Your code should work and be understandable. If that is the case, you need not worry too much about which way is better than the other. I think the way I did it here makes it quite understandable. Line 37, 'if not Accessing_file …' is almost actual, understandable English.

Now that we know the file is accessible, we want to extract a page and read out its binary value. After a long process of trial and error, I found out that one way to do it is to use the *PdfFileMerger*() class of the *PyPDF2* package. A class can bring about an object, in this case a virtual PDF, that only temporarily lives in memory. The functions it has are called methods, which make it possible to bring together different pages from different PDFs. Once satisfied with the virtual PDF, you can then save it to your hard drive. One advantage is that these operations can leave the original files intact. The way we will use it is to bring about one such virtual PDF and, subsequently, only open one PDF and only take one page and add it to the virtual PDF. Line 41 creates such a virtual PDF which at that point consists of zero pages. Another container we need is to take that virtual PDF and read it in its raw binary value, for which we use the *BytesIO*() class. There is no way to do this directly from the virtual PDF as the *PdfFileMerger*() class only offers a few options. The closest we can get is to use the *write* method and instead of writing it to disk, we write it to another virtual file, this time made possible by *BytesIO*().

So, we begin by opening the file with the *open* function. We do this within a *with*-statement which has the familiar punctuation of closing with a colon and indenting the relevant lines of code. The *with*-statement is a *read* and *close* function wrapped into one: once the computer has moved away from the *with*-statement it automatically closes the file to free up memory. We use the *append* method to extract our desired page into the virtual PDF. We then use the *write* method to store this virtual PDF in a virtual file which we call *bytePDF*. Because *bytePDF* is made from the class *BytesIO*(), we have the attribute *getvalue*() available, by which we can create a variable called *page* which then simply has the binary value of the page of the PDF.

Now, we still need to make the step from a PDF page to a JPG image. When we open these PDFs, we see every page as one big photo, but we can easily understand that there is some encoding wrapped around that image; every page of the PDF starts by defining it as a page with its dimensions and other characteristics. The photo is only one part of that page. This means that the binary value of the desired page does not only consist of the binary value of the JPG but also of other things related purely to defining the page as a PDF page. We found out earlier that the binary value of a JPG always starts and ends with a certain value. Our strategy to get the photo out of the page is to search for the first occurrence of JPG, by looking where in the binary value of the page the begin-string of a JPG can be found and where the end. The *find*() method requires a string we want to look for and the position from where we want to begin looking for it. Thus, in the binary value of the PDF page, the JPG starts wherever from position zero onwards the string *b"\xff\xd8"* occurs. Using that place to start looking for the end of the JPG, we now have two variables that contain the beginning position and the end position of the JPG within the PDF page. We can then define a variable *jpgString* by taking the entirety of the page of the PDF and slicing off anything before the start of the JPG and everything after the end of the JPG. We end up with the binary value of the JPG of that PDF page. Since we are convinced that this binary value makes up a valid JPG image, we can simply write it to disk as a .jpg. Wrapping that command within a *with*-statement is again to ensure that we close the file after usage. Python allows to 'open' a file if it does not exist, provided you use the *w*(rite) mode. In that case, Python creates the file for you.

If we would run the script with only what we have discussed so far, nothing would happen. We need to call the *Save_page_PDF* function, and we need to do so only after we have defined it. Putting all functions together in a useful manner is, therefore, done at the end of the script. For example, we could write at the end *Save_page_PDF*(*"test",1*) provided that the Python script file is in the same folder as a PDF called *test.pdf*. The result would be the appearance of a *test.jpg* in the same folder, which would turn out to be the photo of the first page. This, however, would only work if *test.pdf* actually consists of one photo per page. We did not include error handling if this is not the case. In other words, this script is built for a specific situation. To make real use of the computer's power, we can ask it to perform the extraction of a page for hundreds or even thousands of manuscripts. In my case, I had access to 2500 manuscripts spread over a few folders, with each folder containing 500 to a thousand PDFs that all had a predictable name, namely the name of the collection and a number. To make the computer extract the first page of each of them, all that was changing in the path to the file was the number. The easiest way was to use

the second-most used tool of programming (after the *if*-statement), namely the *for*-loop. It has a cousin, too, called the *while*-loop. Both execute code over and over again until told to knock it off. As usual in Python, the grammar and punctuation demand the word 'for', a condition, and then a colon. While the condition is true, the code that is indented is executed. In our case, we simply let a number run from the value 1000 to 2000 using the *range* function. We do so through the variables *startNumber* and *finishNumber*, so that we can easily adjust things. We need the *+1* as, otherwise, number 2000 would not be executed since the *range* function semantically means "up to" and not "up to and including." Often times, the variable that takes each of the values in that range is named *i*. Now, we can use that *i* to construct a thousand different commands of the *Save_page_PDF* function that each target a different PDF file. Notice how we turn the *i*, an integer, into a string to create a full filename that the function will turn into a full file path.

I also included a *print* command in the *for*-loop. This is because when you work with very big PDFs of several hundred megabytes, sometimes more than a gigabyte, the code that we just wrote actually needs longer than a mere moment to digest a PDF like that. Since we loop it over many PDFs, the *print* command gives us some sense of where the script is, which gives a hint about how long it will still take to complete. Additionally, if your script gets stuck on a certain file without giving an error code, this *print* command will let you know which file caused it. If you want to exercise even more control, you could include these few lines of code:

```
# This goes at the top
import time
start_time = time.time()

# This goes at the bottom
print("Finished in %s seconds." % (time.time() - start_time))
```

This code will give you the number of seconds it took to completely execute the script. The *print* command only gives you a line of text in the console, and after doing something else, it is not saved somewhere. *Print* is used mostly for troubleshooting; if you get an error somewhere, you can start *print*ing values of variables at various stages of your script to see how the script behaves and where it goes off the rails. It is useful to accompany this with an *exit*() command which prematurely ends the script. You can place a *print* and *exit* at the beginning of your script and slowly move through it to see around which point the script turns up an error.

6 Step 2: Analysis of an Image

Running the previous script on my data set resulted in a thousand images of covers of manuscripts. Being new at OpenCV, I slowly built my code from the ground up, with continuous testing. The *print*-command, is excellent for building up code and exploring which approach works and which does not. The equivalent of the *print*-command in OpenCV is the *imshow*-function which will display the image you are working on in a window. Your first exercise, then, should be to simply make an object that loads an image through OpenCV using *img = cv2.imread("PathOfFile")*, then to immediately display that image using *cv2.imshow("NameForWindow", img)* (start your code first with *import cv2*). You will additionally need to write at the end *cv2.waitKey(0)*, which tells the computer to keep the window with the image open indefinitely (any other number would mean that amount of milliseconds), and *cv2.destroyAllWindows()* which in combination with the *waitKey*-function will close the window with the image if you press any key.

Getting from those few lines to hundreds of lines of code that actually detect the angle of a flap took me months of coding on and off, trying to push my code in certain directions only to notice that it was not working, slowly crafting it into the final state that you see here. This process was driven by two primary impulses; sometimes I looked up more information about the capabilities of packages, such as *OpenCV* and *NumPy*, and stumbled upon extraordinarily powerful features that seemed to be applicable to my project and used them to device steps that got me closer to my goal; sometimes I brainstormed ways to achieve my goal through a number of intermediate steps and then went to look for techniques to make those steps possible. What can make this a long, painful process is that it requires the fuzziness and multiplicity of humanities research but funneled through the precision and unicity of scientific research. Often, when looking at an image of a codex, you can hardly imagine the precise solution, and when staring at your code, you feel boxed in by the capacities of the technology and wish for a more fuzzy and user-friendly approach. For example, OpenCV has a slew of techniques to identify corners, grouped under the term Feature Detection. Since the angle of the flap can (should?) be calculated by finding the three corner points of the tip and the upper and lower points of the flap, it seemed rather grand to be able to identify those corners with one simple line of code. Trying this out instantaneously made clear to me just how black and white theory is and how vividly colorful (photos of) manuscripts are. The 'corners' that OpenCV was 'detecting' were seemingly random points in the image that had virtually nothing to do with the shape of the codex. My first step was, therefore, to reduce the color image to a black

```
# ----------------------------------------------------------------

# Digital Codicology: Analyzing Islamic manuscripts.
# This code is specifically to measure the angle the cover
flaps make.
# (c) L.W. Cornelis van Lit, O.P., 2017-2018.

# ----------------------------------------------------------------

#Initial requirements.
import cv2
import numpy as np
import os

#User variables
imageStartingNumber = 1091
imageStartingDirection = True
#Either write BW for Black and White processed image to be
displayed, or Color for original to be shown.
imageAppearance = "Color"
maximumEdgePoints = 4
kernelbig = np.ones((10, 10), np.uint8)

# ----------------------------------------------------------------

# This function checks if the file exists and if the file is
incorrupt.
# number is manuscript number as given in the filename.
# Direction is for browsing when encountering a faulty image.
# True means browsing up, False means browsing down.
# Corruption is checked by making sure file size exceeds 100kb.
def Check_image_readable(imageNumber, direction):
  if direction:
    imageNumber = imageNumber + 1
  else:
    imageNumber = imageNumber - 1
  fileName = 'NURUOSMANIYE/NURUOSMANIYE{0}.jpg'.
format(imageNumber)
  if os.path.isfile(fileName):
    fileSize = os.path.getsize(fileName) >> 10
```

and white image in which the flap became clearly visible and distinct from the supporting surface. From there I needed to slowly reduce the number of possible points to one: the tip of the flap, all the while leaving room for exceptions, anomalies, and irregularities.

In this code, we will need three packages, and so we import them first. Next, we set up some variables that determine the initial image that should be analyzed. When cleaning up my code, I split it into four functions, each covering a specific part of what the computer needed to do. We first check if the file is an actual and complete image, then we analyze the image to get to the flap, then analyze the flap to get to the angle, and last we want to display our results so that we can get visual confirmation.

6.1 *The Function* Check_image_readable

This function makes use of two observations. One is that all our files have the same name except for a number, the other is that files under about a 100kb are corrupt. We do not want to perform an analysis on a corrupt image. Instead, we would like the computer to move over to the next image. If we would run the angle detection code on only one image per turn, we could have chosen to simply give back an alert when an image is corrupt instead of moving over to the next image. In this case, we wish to see how the script performs under multiple images since there is a lot of variation in the photos and manuscripts, and running it on only one image would possibly keep us blind to edge cases that our code does not facilitate. We use a variable *direction* to indicate which way the computer should cycle; to a number higher or a number lower. Here, I have chosen to do so by making *direction* either True or False. The *if*-statement speaks for itself: 'if direction is true', then evaluate the manuscript whose filename is one number higher; 'if direction is false', then evaluate the manuscript with one number lower. Next, we establish the path to the file by the variable *fileName*. As you may notice, I have hard-coded the path to the file, since this script is in first instance meant to test our setup. To productively make use of this script, we would do well to make this more dynamic by defining *fileName* from the combination of several variables defined at the beginning, as we did in the previous script. In a next *if*-statement, we first check if there even is a file like we just defined, using the *os* package. I chose this function over the *open*-function since it gives us an obvious *True/False* dichotomy and since we will use the *os* package anyway in order to gauge the file size of that file using *getsize*. This *getsize* command gives an answer in bytes, which we can change to kilobytes by writing *>> 10*. Then, we evaluate that file size: if it is over 100, we let the computer go on to the next function; if it is not, we let the computer start over from the beginning of *Check_image_readable*. This would not

```
    if fileSize > 100:
      Analyze_image(imageNumber)
    else:
      Check_image_readable(imageNumber,direction)
  else:
    Check_image_readable(imageNumber,direction)

# ----------------------------------------------------------------

#This function analyses the image.
def Analyze_image(imageNumber):
  img = cv2.imread('NURUOSMANIYE/NURUOSMANIYE{0}.jpg'.
format(imageNumber))
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  small = cv2.resize(img, (0, 0), fx=0.8, fy=0.8)
  smaller = cv2.resize(small, (0, 0), fx=0.5, fy=0.5)
  originalImage = smaller
  height, width, _ = originalImage.shape

  ret, thresh1 = cv2.threshold(gray, 180, 255, cv2.
THRESH_BINARY_INV)
  scaled = cv2.resize(thresh1, (0, 0), fx=0.8, fy=0.8)
  openimg = cv2.morphologyEx(scaled, cv2.MORPH_OPEN, kernelbig)
  ret, thresh2 = cv2.threshold(openimg, 1, 255, cv2.
THRESH_BINARY)
  scaledagain = cv2.resize(thresh2, (0, 0), fx=0.5, fy=0.5)

  _, contours, _ = cv2.findContours(scaledagain, cv2.RETR_
EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
  cv2.drawContours(originalImage, contours, -1, (0, 255, 0), 1)
  mainContour = max(contours, key=len)

  hull = cv2.convexHull(mainContour, returnPoints=False)

  hullContourX = []
  hullContourY = []
  for indexHull in hull:
    hullContourX.append(mainContour[indexHull[0]][0][0])
    hullContourY.append(mainContour[indexHull[0]][0][1])
```

result in analyzing the same image again, since we change the *imageNumber* at the beginning. Thus, if the computer hits a corrupt image, it will cycle upward or downward, depending on the direction, long enough to find a good image. Notice that the script does not handle the case in which the computer would cycle out of the range of manuscripts. For example, if in the folder there are files with MANUSCRIPTNAME500 to MANUSCRIPTNAME1000, if the computer goes to 499 or 1001, it would theoretically search for a valid image indefinitely. In practice, it will produce an error message stating that the maximum recursion depth has been exceeded. We did not include guard rails for these cases as this script relies on the user to not enter those error states but instead stay within the boundaries of the files.

6.2 *The Function* Analyze_image

With the knowledge that the image is valid, we now begin by reading in the image. Many of the variable names in this function are short and not terribly descriptive, and that is because their use is limited to a specific place in the script; if you understand the context, you understand the meaning of the variable. In this function, the first step is to turn the image into a greyscale image. We also wish to preserve a copy of the original for display purposes. Since the images I worked with are way too big to display fully on a screen, the script includes commands to resize them. Since the resizing of the greyscale image is done in two stages and may have an impact on the process of making the manuscript white and the background black, I resize the original as well in two steps. This may be a part of the script that is ripe for improvement, but in here lies a lesson for us: although a clean and lean script enhances its usability, there comes a point after which cleaning up does not give us that much more usability that it merits the time investment. In this function, after the resizing, we then want to know how big the image has become. Of the different ways there are to get to this information, we use here *NumPy*'s *shape* command which returns the height, width, and number of channels. To store each of them separately, we define three variables at the same time, separated by a comma. Since we only require the first two, we can forget about the last one by inserting an underscore.

The next few lines of code transform the image into a black background and a white codex. This is to some extent an art, not a science. This means that to some extent, you are simply left to play around with some of the available tools and their settings until you get the desired result. Furthermore, this also depends on the images you have. With the test data used here, a particular difficulty presented itself in that the surface on which the manuscripts were lying had a prominent pattern and was relatively close in color to some of the

```
#Hand over output to function that displays image
  Display_image(imageNumber, originalImage, scaledagain,
hullContourX, hullContourY)

# ----------------------------------------------------------------

def Find_angle(hullContourX, hullContourY, imageWidth,
imageHeight):
  # Using NumPy requires arrays
  reducedXArray = np.array(hullContourX)
  reducedYArray = np.array(hullContourY)

  # Using NumPy to get index numbers of all Y values within
15% of middle
  reducedYLength = len(hullContourY)
  varianceY = imageHeight * 0.15
  minimumY = imageHeight / 2 - varianceY
  maximumY = imageHeight / 2 + varianceY
  decideMiddlePoints = np.zeros(reducedYLength, dtype=np.int)

  for correctYvalue in range(0, reducedYLength):
    if minimumY <= hullContourY[correctYvalue] <= maximumY:
      np.put(decideMiddlePoints, correctYvalue, 1)
      xpos = hullContourX[correctYvalue]
      ypos = hullContourY[correctYvalue]

      # Ruling out spurious points on non-flap side by
checking if nearby points in terms of x-value are near edge
in terms of y-value (i.e. in a corner)
      # Also ruling out flaps with distortions, i.e. deleting
all middle points on the left/right side of the image where
there is a point which meets above requirement.
      for otherXvalues in range(0, reducedYLength):
        if (xpos - 5 <= hullContourX[otherXvalues] <= xpos
+ 5) and (hullContourY[otherXvalues] < imageHeight * 0.2 or
hullContourY[otherXvalues] > imageHeight * 0.8):
          if hullContourX[otherXvalues] <= imageWidth/2:
            np.put(decideMiddlePoints,np.where(reducedXArray
<imageWidth/2),0)
```

codices. With a clean black background, things will obviously be much easier. *Threshold* is a useful function which looks for all pixels with a value higher than a certain limit and gives all these pixels the same value. To remove noise and make the silhouette of the codex smoother, we can use a morphological function called *opening*. This made it necessary to threshold again, but this time to catch every non-pure black part and make it pure white. For a reason that I did not look deeper into, after a threshold, I first needed to scale the object even if only to the same scale in order for other functions to operate on the object. This is an example of finding a solution to a problem that seems to run way too deep for our level of expertise: if a certain band-aid works to counter the symptoms of a problem, then it might be alright to actually use that band-aid, as the actual cause of the problem might be too complicated to solve or even to understand. Even if the root cause was fairly easy to figure out and correct, there is not much lost here to use a *scale* function merely to get things to work. Of course, there is a small-time penalty when executing the code, but we 'only' work with thousands of files and do not need a super-powered mainframe to crunch our numbers. Even if our own computer will have to crunch an extra second per manuscript (it won't), that should be perfectly acceptable to us.

Now that we have the manuscript as a white object, we can ask OpenCV to find the contours of all white objects. If you experiment with the scripts as you are reading this, you will notice that some manuscripts will not be one solid white blob but have black spots and sometimes isolated small white spots within the black spots. In other words, some parts of the manuscript have been effaced and we will have to take this into account as we move ahead. You may have also been puzzled as to why we created the objects white and the background black since our human eyes would recognize objects much better if the object(s) are black on a white surface. This is not so for OpenCV. The *findContours*-function, which we use, finds white objects on a black background. On the next line of code, we command the computer to draw the contours in bright green. I made a promise earlier to keep functionally different parts separated which would mean that, in this function, we only analyze the image while later taking care of displaying the image. You could argue, therefore, that this is an imperfection in the code. I kept it in to show you my workflow: while developing this, I followed the *drawContours*-function with an *imshow*-command to display the image and check to see if the code generated a desirable outcome.

Since there can be multiple contours, I assume that the largest contour must be the contour of the actual codex. This assumption will produce a couple of so-called type II errors, i.e., false negatives: for images where the flap has become its own white blob, the computer will fail to find a flap (even though it

```
            else:
              np.put(decideMiddlePoints, np.where(reducedXArray
> imageWidth / 2), 0)
            break

  # Only proceed if a flap can be found.
  # Get X,Y value of tip-point, making sure there are some
points that could be the tip

  if not np.count_nonzero(decideMiddlePoints) == 0:
    # Get X,Y value of only tip-points
    tipOfFlapYArray = reducedYArray[decideMiddlePoints != 0]
    tipOfFlapXArray = reducedXArray[decideMiddlePoints != 0]
    restOfHullXArray = reducedXArray[decideMiddlePoints == 0]
    restOfHullYArray = reducedYArray[decideMiddlePoints == 0]

    # tipX needs to be decided, left or right
    # Is the tip on the left?
    if np.average(tipOfFlapXArray) < imageWidth / 2:
      # Get X,Y value of furthest tip-point
      reducedXMinimumArray = np.where(tipOfFlapXArray ==
tipOfFlapXArray.min())
      reducedXMinimumList = reducedXMinimumArray[0].tolist()
      tipX = int(tipOfFlapXArray.min())

      # tipY needs to be estimated as an average
      tipY = 0
      for tipCoordinate in reducedXMinimumList:
        tipY = tipY + tipOfFlapYArray[tipCoordinate]
      tipY = int(tipY / len(reducedXMinimumList))
      # Draw the tip-point
      # cv2.circle(imageForDisplay, (tipX, tipY), 6, (0, 50,
250), -1)
    # Is the tip on the right?
    elif np.average(tipOfFlapXArray) > imageWidth / 2:
      reducedXMinimumArray = np.where(tipOfFlapXArray ==
tipOfFlapXArray.max())
      reducedXMinimumList = reducedXMinimumArray[0].tolist()
      tipX = int(tipOfFlapXArray.max())
```

is there). Since we are working with large numbers, false negatives are always more desirable than false positives. Err on the side of caution.

With the main contour of the codex found and defined as five hundred to a thousand points, our hunt for the three points defining the tip and both edges of the flap has drawn closer. In one swoop, we can reduce this to about twenty to forty by letting OpenCV draw a convex hull around the contour. A convex hull is the shape that encapsulates the entire codex in the least possible points. It is not the surface area that is minimized, but the number of corners. So, within the hull, there may be some 'empty space' between the border of the hull and the border of the contour. In the case of Nuruosmaniye 1091, the entire photo consists of just more than 4 million points, the contour of the codex consists of 782 points, whereas the hull consists of 23 points. It is now our job to find the three points of those 23 that define the flap, or to begin with just the one that defines the tip.

Before we go into that, we let the *Analyze_image* function do one more thing and that is to store the X and Y coordinates of the hull points in lists. A list is much like a vector or an array: one object that contains several values in an ordered manner; much like a bus when looked from the side, behind every window sits somebody (or nobody). They all sit in the same bus, all have a specific place, and yet each is different. When we speak of variables, we usually mean a word used as a name that stores only one value (a string, an integer, or boolean), such as *name = "Cornelis"*, or *publicationYear = 2020*, or *published = True*. With lists, we start to group together a bunch of such values into one object. The definition is the same as single variables; we simply type a word (the name of the object) followed by an equal sign followed by the data we want that word to hold, such as: *divisionOfBook* = [*"Introduction", "Theory", "Practice", "Conclusion", "Postscript"*]. Everything between the square brackets now belongs to *divisionOfBook*. A list is only one type of such collection of values, and the general term for such types of objects is 'data structure.' This technical term can be helpful to understand what is going on: the data is a number of values, and the structure is the way it is stored. For a list, then, the structure is simple: all values are stored one after the other, and the order in which they are stored is preserved. This order-preservation is very useful, because it means that in Python, the value of a specific place can be obtained by calling the name of the object and using an index number with square brackets, such as *divisionOfBook*[*1*], which will return "Theory" (note that Python starts counting at zero.)

When we used the *convexHull*-function we actually got back the index of each point of the contour that together makes up the convex hull. So, with that index and the contour, we can extract the X-values and Y-values. First, we

```
        # tipY needs to be estimated as an average
        tipY = 0
        for tipCoordinate in reducedXMinimumList:
          tipY = tipY + tipOfFlapYArray[tipCoordinate]
        tipY = int(tipY / len(reducedXMinimumList))
        # Draw the tip-point
        # cv2.circle(imageForDisplay, (tipX, tipY), 6, (0, 50,
250), -1)
      else:
        #print("Middle points undeterminedly left or right.")
# It is a bit hacky but we need to specify all seven return
values as None. If we decide later to add more return values,
more 'None' values need to be added here
        return (None, None, None, None, None, None, None)
    else:
      #print("No middle points found!")
      return (None, None, None, None, None, None, None)

    # Check if hull is formed correctly
    if (abs(restOfHullXArray - tipX) < 4).any():
      print("Some point of hull too close to X value of tip.
Computer likely did not identify hull correctly.")
      return (None, tipX, tipY, None, None, None, None)

    # Now find points along the edge and calculate angle

    tipUpperX = tipX
    tipUpperY = np.min(tipOfFlapYArray[reducedXMinimumList])
    tipLowerX = tipX
    tipLowerY = np.max(tipOfFlapYArray[reducedXMinimumList])

    #In the case of e.g. NURU1575 the middle-point-finder could
only find one point, so we need to search by hand for the
other tip-point.
    if len(decideMiddlePoints[decideMiddlePoints==1]) == 1:
      decideOtherTip = np.where(imageWidth - reducedXArray < 4)
[0].tolist()
      if decideOtherTip:
        tipUpperY = np.min(reducedYArray[decideOtherTip])
```

simply make new, empty lists called *hullContourX* and *hullContourY*. Then, we loop through the indices that the *convexHull*-function created and take from the contour of the codex the X-value and the Y value. The *hull* is wrapped twice in a list, meaning that each index is not stored as an integer but as a list containing one element which is an integer. So to get to the correct position in the list of *mainContour*, we should not call *mainContour*[*ElementInHull*], but *mainContour*[*ElementInHull*[*0*]] instead. In Python, counting starts at zero; so, the first element in a list can be reached by [0]. For *mainCountour*, the same is true; each point is not just two values representing the X and Y coordinates, but each point has a list in which there is one element, namely a list of two values. In this manner, we need to add [0] just to get to the X,Y-value pair. Since X is first in the list and Y second, we can get to X by again adding [0], and we get to Y by adding [1]. We use the *append*-method to add those values to the *hullContourX* and *hullContourY* lists. To understand these last couple of technical moves, it might be worthwhile to read the code and the explanation again, carefully, and to play around with it yourself by placing a few *print* commands and an *exit* command right after and then running the script. With that, this function has played its part and needs to hand over its findings to the next function. Of course, we should reasonably expect this to be the *Find_angle* function. However, in the course of developing this code, I came to enjoy it better to immediately parse it to *Display_image* and let that function call upon *Find_angle*.

6.3 *The Function* Display_image

This function is all about visually presenting the information. In the end, we will be able to display the original image, a contour of the codex, the points that define the hull, the tip and the corners of the flap, and a message telling us the angle. I have added a rudimentary user interface in which pressing the key E makes one switch the view from the original manuscript photo to the white-on-black image as processed by OpenCV. Additionally, pressing the key Q makes one switch to the photo one number down, and pressing the key W will switch to the photo one number up.

The first task is perhaps counter-intuitive: to make a color image out of a greyscale image. This does not mean that the photo itself turns colorful again, it remains a greyscale image (or rather, white on black now that it has undergone certain transformations). But it simply means we tell OpenCV we want three channels per pixel again instead of one. This is needed so that we can draw on the image in color, such as the points that define the hull. Were we to draw those points on a greyscale image, they would be grey too and they

```
      tipLowerY = np.max(reducedYArray[decideOtherTip])
      restOfHullXArray = np.delete(restOfHullXArray,decideOth
erTip)
      restOfHullYArray = np.delete(restOfHullYArray,decideOth
erTip)

  #For left flap we need to look at minimal X values, for
right flap maximal X values.
  if tipX < imageWidth / 2:
    # Get all XY that are in the upper quadrant
    upperRestOfHullXArray = restOfHullXArray[restOfHullYArray
< imageHeight / 2]
    upperRestOfHullYArray = restOfHullYArray[restOfHullYArray
< imageHeight / 2]

    upperRestOfHullYArray = upperRestOfHullYArray[upperRestOf
HullXArray < imageWidth * 0.2]
    upperRestOfHullXArray = upperRestOfHullXArray[upperRestOf
HullXArray < imageWidth * 0.2]

    # Get all XY that are in the lower quadrant
    lowerRestOfHullXArray = restOfHullXArray[restOfHullYArray >
imageHeight / 2]
    lowerRestOfHullYArray = restOfHullYArray[restOfHullYArray >
imageHeight / 2]

    lowerRestOfHullYArray = lowerRestOfHullYArray[lowerRestOf
HullXArray < imageWidth * 0.2]
    lowerRestOfHullXArray = lowerRestOfHullXArray[lowerRestOf
HullXArray < imageWidth * 0.2]

    def GetUpperPointNumber():
      return((np.where(upperRestOfHullXArray ==
upperRestOfHullXArray.min())[0]).tolist())

    def GetLowerPointNumber():
      return((np.where(lowerRestOfHullXArray ==
lowerRestOfHullXArray.min())[0]).tolist())
  else:
    # Get all XY that are in the upper quadrant
```

would not visually stand out. We also calculate the height and width. We had done so before, but variables defined within a function only live within that function and cannot be accessed outside of it. We could have parsed the values into this *Display_image* function, but recalculating them seemed just as convenient.

In the next block of code, we encounter one way that a variable is able to be accessed in places other than where it is locally defined, and that is to put *global* in front of it. I did this so that *imageAppearance* would be recognized wherever. The *if-else*-statement speaks for itself: it looks at the variable defined at the very beginning and either makes *imageAppearance* the image processed by OpenCV (white codex on black background) or the image as it originally can be found on your hard drive. If the variable is set to anything else, the code will not execute, and a custom error message is printed to the console.

In the next block, we use a *for*-loop to make all the corners of the hull visible. A *for*-loop is convenient to write these commands in an easy, compact form and also has the added benefit that it is dynamic: whether the hull has 20 points for one image or 24 points for another image, it will iterate over all the points. I figured the corners can be best made visible by points. So, we will draw very small circles which we can do with the aptly termed *circle*-function of OpenCV. This functions needs to know the image it should draw on, the X,Y coordinates (counting from the top-left corner) of the center of the circle, the radius of the circle, the color, and the thickness of the outline. Using –1 as the thickness tells OpenCV that the circle should not have an outline but should have a solid fill. For every point of the hull, we repurpose the variable *xPos* and *yPos* to set the X and Y position of that point, and then draw a circle on it. I made an index on the fly, starting from zero and ending in whatever the length of either *hullContourX* or *hullContourY* is (they are obviously the same length). I am sure there are other ways to go about drawing point-like circles for each corner of the hull, but I found this approach simple (only four lines of code) and understandable.

Next, the function *Find_angle* is called. This function spits out several answers: the angle, the X and Y coordinates of the tip of the flap, and the X and Y coordinates of points along the upper edge of the flap and the lower edge of the flap, which are most often the actual corners of the flap. All these answers need to be filled as a value of a variable within the function of *Display_image*. We can simply do so by defining these variables as the function, separated by commas, of course, according to the order of the answers that the function gives back. For the function to do its magic, it needs four pieces of information: the corner points of the hull in its X and Y coordinates, the width of the

```
    upperRestOfHullXArray = restOfHullXArray[restOfHullYArray
< imageHeight / 2]
    upperRestOfHullYArray = restOfHullYArray[restOfHullYArray
< imageHeight / 2]

    upperRestOfHullYArray = upperRestOfHullYArray[upperRestOf
HullXArray > imageWidth * 0.8]
    upperRestOfHullXArray = upperRestOfHullXArray[upperRestOf
HullXArray > imageWidth * 0.8]

    # Get all XY that are in the lower quadrant
    lowerRestOfHullXArray = restOfHullXArray[restOfHullYArray
> imageHeight / 2]
    lowerRestOfHullYArray = restOfHullYArray[restOfHullYArray
> imageHeight / 2]

    lowerRestOfHullYArray = lowerRestOfHullYArray[lowerRestOf
HullXArray > imageWidth * 0.8]
    lowerRestOfHullXArray = lowerRestOfHullXArray[lowerRestOf
HullXArray > imageWidth * 0.8]

    def GetUpperPointNumber():
      return((np.where(upperRestOfHullXArray ==
upperRestOfHullXArray.max())[0]).tolist())
    def GetLowerPointNumber():
      return((np.where(lowerRestOfHullXArray ==
lowerRestOfHullXArray.max())[0]).tolist())

  # Establishing upper edge points
  upperEdgePoint = []
  upperEdgeX = []
  upperEdgeY = []
  for pointNumber in range(0, maximumEdgePoints):
    if pointNumber < len(upperRestOfHullXArray):
      upperEdgePoint.append( GetUpperPointNumber()  )
      upperEdgeX.append( upperRestOfHullXArray[upperEdgePoint
[pointNumber][0]] )
      upperEdgeY.append(upperRestOfHullYArray[upperEdgePoint
[pointNumber][0]])
```

image and the height of the image (that is, not the original image on file but the image as it is processed by OpenCV). We will discuss the *Find_angle* function later.

Then, the first thing to check is if an angle was found. If no angle was found, the message we want to display is that no flap was detected. If an angle was detected, we want to set the variable *message* to that. I used a feature of Python to include a variable in a string, by placing curly brackets around an index number (starting with zero) and by following the string with *.format*(*VariableName*). If there is an angle, we also want to make the tip and the edges of the flap visible; so we draw points, this time bigger and in other colors than when we drew points for the corners of the hull.

Next, we want to display the *message*. We do this outside of the *if*-statement because we do this no matter the outcome. If you observe closely, you will see that I have OpenCV draw the same message thrice. The first two times, the message is black and the third time, it is purple-pink. I do this so that the purple-pink text looks like it has a black border around it which enhances the readability.

All that is left to do is give OpenCV the command to display the image on our monitor, followed by a command to freeze it until a keystroke. Any key other than Q, W, or E will close the window. By evaluating the keystroke using *ord*, we can make the code readable for humans, as it now simply reads 'if [the] key [stroke] is q …'. When we press Q on our keyboard, we want the computer to analyze the manuscript whose number is one less than the current one. The easiest way is to close the current window and let the script run all over again but this time with *direction* set to *False*, that is, 'downward'. For W, we do the same but with *direction* set to *True* or 'upward.' For pressing E, we want to switch between normal view and analyzed view, which we can do by only letting the last function, *Display_image*, run again but with the variable *imageAppearance* changed. At the moment, we only have two different views, normal and analyzed. So we could have made *imageAppearance* a boolean variable, switching between *True* and *False*. For readability and for allowing future expansion into other views, I thought it wiser to have *imageAppearance* be a string which should be either 'BW' or 'Color.'

The script finally ends with a call to the function *Check_image_readable*, to set everything in motion. I placed a minus one after *imageStartingNumber* since *direction* is at first set to *True* and will, therefore, increase the *imageNumber* by one in the *Check_image_readable* function. All that is left to discuss for this script is the function that figures out the angle based on the corner points of the hull.

```
      upperRestOfHullXArray = np.delete(upperRestOfHullXArray,
upperEdgePoint[pointNumber])
      upperRestOfHullYArray = np.delete(upperRestOfHullYArray,
upperEdgePoint[pointNumber])

  # Establishing lower edge points
  lowerEdgePoint = []
  lowerEdgeX = []
  lowerEdgeY = []
  for pointNumber in range(0, maximumEdgePoints):
    if pointNumber < len(lowerRestOfHullXArray):
      lowerEdgePoint.append( GetLowerPointNumber() )
      lowerEdgeX.append(lowerRestOfHullXArray[lowerEdgePoint[
pointNumber][0]])
      lowerEdgeY.append(lowerRestOfHullYArray[lowerEdgePoint[
pointNumber][0]])
      lowerRestOfHullXArray = np.delete(lowerRestOfHullXArray,
lowerEdgePoint[pointNumber])
      lowerRestOfHullYArray = np.delete(lowerRestOfHullYArray,
lowerEdgePoint[pointNumber])

  # !!! Note that upper/lowerRestOfHull are now empty !!!

  # Calculate degree of angle based on lower edge points
  # First apply Theorem of Pythogaras to find all lengths,
with A = horizontal, B = vertical, C = diagonal

  upperAngles = []

  for pointNumber in range(0,len(upperEdgeX)):
    # First apply Theorem of Pythogaras to find all lengths,
with A = horizontal, B = vertical, C = diagonal
    A = abs(tipUpperX - upperEdgeX[pointNumber])
    B = abs(tipUpperY - upperEdgeY[pointNumber])
    C = A**2 + B**2
    # Then use Law of Cosine to find angle. Answer is
returned
in Radians so it needs to be transposed to Degrees for human
readability.
```

6.4 *The Function* Find_angle

We have reduced our humanities problem of finding out more about the flap to a mathematical problem of finding a triangle based on a couple of dozen coordinates. What we need to do, then, is figure out regularities that we can rely on. As it turns out, many things are unreliable in the case of these digitized manuscripts. The images have different sizes. There is no guarantee of a flap. The number of points needed to define the hull is different for each photo. The flap might be left or right. The codex might not be exactly straight in the photo. Sometimes the tip is cut out of the picture, and so on. Sometimes, you think you can exploit a regularity. But then as you cycle through a few manuscripts, you come to one with an exception and the computer does something unexpected or it outright crashes.

After several unsuccessful attempts, I landed on the strategy to find the tip of the flap by looking only at points of the hull that were within the middle 30% height-wise. Since we do a fair number of mathematical operations in this function, we need to rely on the *NumPy* package. Moreover, for that to work best we need to convert our list to NumPy arrays. We establish a bandwidth of Y values and make an array that has as many zeros as there are coordinates. This array, *decideMiddlePoints*, will be filled with the value 1 in every place where the Y value of a coordinate is within the bandwidth. This will be useful because we can exploit the fact that multiplying by one preserves the value and multiplying by zero is zero. Thus, we can multiply the array with all the X values with this *decideMiddlePoints*. We do the same for the array with the Y values and, then, we have arrays that only have values for points that fall in that bandwidth, while still maintaining the original length. The rest are all zeros since multiplying by zero is zero. If you are already starting to lose your grip on what we are attempting to do here, a good way to once again understand what is happening is to run the script on an example image and print out all kinds of variables at various points (and include an *exit*() command right after to stop the script) to see how the variables are filled with values. Even better is to tinker with the formulas and see what kind of effect that has on the values in the variables.

Using a *for*-loop, we cycle through all values of the array that contains the Y-values of the corners of the hull, and we check to see if that Y value is in the middle 30% bandwidth. If so, we place a 1 in that position in the *decideMiddlePoints*. Subsequently, we perform another check. If there is another point that is less than (or equal to) 5 pixels away X-wise, and the corresponding Y value for that X value is within 20% of the top of the image or of the bottom, we know that wherever this middle point is, it cannot be a part of the flap. It could only

```
    upperAngles.append( np.rad2deg(np.arccos((A**2 + C - B**2) /
(2 * A * np.sqrt(C)))) )

  lowerAngles = []

  for pointNumber in range(0, len(lowerEdgeX)):
    A = abs(tipLowerX - lowerEdgeX[pointNumber])
    #A is 0 which is causing the problem. Not allowed to
divide by 0.
    B = abs(tipLowerY - lowerEdgeY[pointNumber])
    C = A**2 + B**2
    lowerAngles.append(np.rad2deg(np.arccos((A ** 2 + C - B
** 2) / (2 * A * np.sqrt(C)))))

  # Entire angle, rounded to zero decimals (would give false
sense of accuracy otherwise) is:
  if not upperAngles or not lowerAngles:
    print("Error calculating angle!")
    return (None, None, None, None, None, None, None)
  else:
    finalAngle = int(np.rint( np.average(np.
asarray(lowerAngles)) + np.average(np.asarray(upperAngles)) ))
    return(finalAngle, tipX, tipY, lowerEdgeX[-1],
lowerEdgeY[-1], upperEdgeX[-1], upperEdgeY[-1])

# --------------------------------------------------------------

def Display_image(imageNumber, original, analyzed,
hullContourX, hullContourY):
# Convert the analyzed result from BW to Color so that we can
draw in color over it
  analyzedConverted = cv2.cvtColor(analyzed,cv2.
COLOR_GRAY2BGR)
  imageHeight, imageWidth, _ = analyzedConverted.shape

# Do you want to see the original or the processed image?
  global imageAppearance
  if imageAppearance == "BW":
    imageForDisplay = analyzedConverted
  elif imageAppearance == "Color":
```

be part of a non-flap side, or perhaps the computer was unable to distinguish the codex from the background. So, we not only rule out that middle point, we also rule out all points on that side of the image. With side I mean here the left from the middle or right from the middle. If we did indeed find such a point adjacent to a middle point, then we just make *decideMiddlePoints* zero for all points, middle points or not, on that side of the image, and we no longer need to look for adjacent points; hence, the *break* command, which prematurely ends the *for*-loop.

We should be left with an array *decideMiddlePoints* that has the value 1 in each place where the corner points of the hull have something to do with the tip of the flap. If *decideMiddlePoints* only has zeros, the computer could not detect a flap. If there are some ones, we now split up the corner points of the hull into those that have to do with the tip of the flap and those that do not. We can do so very easily in Python by using a condition within the index. Written out, line 112 says the following: get the Y values of all points that are close to the tip of the flap by taking only those Y values of the points that define the hull around the white blob (which is the codex) where the corresponding value of *decideMiddlePoints* is not zero.

We now have one or more candidates for the tip of the flap. Since the tip is the point that sticks out the most, to find the tip can now be done in two steps. First, we check on which side of the image the tip-points are. If they are on the left of the middle, the tip-point must be the point which has the smallest X-value, which can be found by *.min*(), otherwise we can use *.max*(). If the flap extends outside the image, the tip itself is not visible and two points would be found, for which both points either have X = 0 or X = width of image. The Y value can then be found as an average of the two points. This *tipY* is only used for visualization and not for angle calculation.

The next block of code, I am not entirely sure about. I encountered some cases in which the computer produced a strange result or no result at all but an error, and it seemed I could step over those cases by including a check if no other point of the hull is right next to tip, within 4 pixels distance. This check is only done on points of the hull that are out of the 30% bandwidth. So instead of trying to find the angle in these cases with more effort, I simply discard them as cases in which the computer could not find the angle. This strategy relies on enough cases to succeed that a slightly bigger loss is of no concern. If your use case cannot afford such a loss, you would have to invest more time to research what is going on in these unusual cases and what can be done to measure the angle nonetheless.

```
    imageForDisplay = original
  else:
    print("Variable imageAppearance must be set to BW or
Color!")
    exit()

  #Drawing all points of the outer, reduced hull
  for correctYvalue in range(0,len(hullContourX)):
      xpos = hullContourX[correctYvalue]
      ypos = hullContourY[correctYvalue]
      cv2.circle(imageForDisplay, (xpos, ypos), 3, (255, 0, 0),
-1)

  #Calling Angle finding function
  message, tipX, tipY, lowerEdgeX, lowerEdgeY, upperEdgeX,
upperEdgeY = Find_angle(hullContourX,hullContourY,imageWidth,
imageHeight)
  if message == None:
    message = "No Flap Detectable!"
  else:
    message = "Angle of flap is {0} degrees.".format(message)
    cv2.circle(imageForDisplay, (tipX, tipY), 9, (0, 50,
250), -1)
    cv2.circle(imageForDisplay, (lowerEdgeX, lowerEdgeY), 6,
(0, 250, 250), -1)
    cv2.circle(imageForDisplay, (upperEdgeX, upperEdgeY), 6,
(0, 250, 250), -1)

# Putting text and showing image
  cv2.putText(imageForDisplay,str(message),(int(imageWidth/2 -
139), int(imageHeight/2 + 1)),cv2.FONT_HERSHEY_DUPLEX, 1, (0, 0,
0), 1)
  cv2.putText(imageForDisplay,str(message),(int(imageWidth/2 -
140), int(imageHeight/2)),cv2.FONT_HERSHEY_DUPLEX, 1, (0, 0,
0), 1)
  cv2.putText(imageForDisplay,str(message),(int(imageWidth/2 -
141), int(imageHeight/2 -1)),cv2.FONT_HERSHEY_DUPLEX, 1,
(120, 0, 220), 1)
  cv2.imshow("{0}".format(imageNumber), imageForDisplay)
```

6.5 *The Mathematics behind Finding the Angle*

The rest of the code relies on the idea that the angle of the flap can be found if you add up the angle of the line that goes horizontally straight through the tip and the line from the tip along the upper edge of the flap, and the angle with the line from the tip along the lower edge of the flap. Let us do some back-of-the-envelope mathematics to see how that works. From there, we can write in plain English a pseudo-code of what we need to do, and then we can convert that into Python and NumPy code.

Following the figure below, we can see how we can split the entire angle of the flap into two angles, one top, one bottom. Further, according to the theories of similar triangles, it does not matter where we find the points to construct the angle. If part of the flap is cut off from view, we simply calculate not from the tip but from the first visible point, and we use a horizontal line through that point. Similarly, again, according to what we know of similar triangles, we do not need to know the actual topmost and bottommost corner point of the flap, as any point along the edge of the flap will do. Last, if the photo of the manuscript is a bit skewed, this does not matter, since the loss of angle on one side is made up by what is gained on the other side.

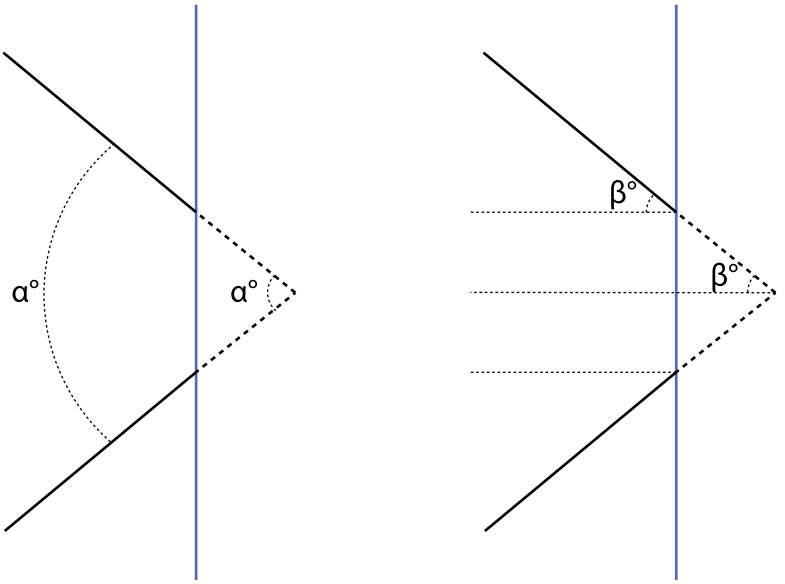

FIGURE 7.1 Proof that we can measure the top and bottom angle to obtain the entire angle of the flap

```
# Awaiting keyboard input.
# The key ‘q’ will show the manuscript with call number one
less than current.
# The key ‘w’ will show MS with call nr. one more.
# Any other key exits the program.
  key = cv2.waitKey(0) & 0xFF

  if key == ord(“q”):
    cv2.destroyAllWindows()
    Check_image_readable(imageNumber, False)
  elif key == ord(“w”):
    cv2.destroyAllWindows()
    Check_image_readable(imageNumber, True)
  elif key == ord(“e”):
    if imageAppearance == “Color”:
      imageAppearance = “BW”
      Display_image(imageNumber, original, analyzed,
hullContourX, hullContourY)
    else:
      imageAppearance = “Color”
      Display_image(imageNumber, original, analyzed,
hullContourX, hullContourY)
  else:
    cv2.destroyAllWindows()

# ----------------------------------------------------------------

# Execute the code by starting the first function.
Check_image_readable(imageStartingNumber-1,
imageStartingDirection)
```

We have enough information if we can find the coordinates for two points along the (upper or bottom) edge of the flap. By taking the absolute value of subtracting their Y values, we get the 'height' of the triangle. By taking the absolute value of subtracting their X values, we get the 'width' of the triangle. Now, we have constructed a right triangle (see the figure) for which the theorem of Pythagoras counts, which allows us to calculate the length of the flap side of the triangle.

With the length of all three sides known, we can use the law of cosines which works on any triangle. This law says that the following formula is true, with a, b, and c the length of the edges and y the angle of a corner: $c^2 = a^2 + b^2 - 2ab \cos y$. This formula expresses the length of one side into the lengths of the other sides and the cosine of the angle of the opposite corner. We can flip the formula to express the cosine of the angle into the lengths of the triangle as such: $\cos y = (a^2 + b^2 - c^2) / 2ab$. To get to the angle y, we then need to take the inverse of the cosine, which is called arccos. We do not need to know much more about what *cos* and *arccos* really are since we will let NumPy do the calculations for us.

In other words, we need to find any two points along the upper edge of the flap to calculate the upper angle, do the same for the lower edge of the flap. Then, by putting them together, we get the angle of the flap. To be sure and accurate about this, we shall tell the computer to find a couple of points along each edge and take the average of the angles. We will likely get an answer with many decimals, but this is a false sense of accuracy. To do justice to our method, we shall round off the answer to whole degrees, as I think no greater degree of accuracy may be expected.

Getting back to our code, we first establish the tip points, which may be just one if the tip of the flap is visible on the image. But it may be two if the tip falls out of view. As we proved above, the two points can stand in for the tip while calculating the angle of the flap.

We need to distinguish two cases: if the flap is left or if the flap is right. If the flap is left, we distinguish two further cases: the points along the upper edge and the points along the lower edge. Looking at the upper edge case, we find the points in two steps: first, we find all points on the upper side of the image, then among those points we find all points that are within 20% of the left side of the image. By including that condition within the index, we can write our code in a very compact manner. Notice that the order is important here: we reduce our coordinates to the upper half by first doing so for X values, then Y; for the reduction to the left quadrant we first do so for Y values, then X. Otherwise, the index numbers would not be accurate. As a last act, we define two functions

we will be using later on. By defining them here, we keep the number of lines of code to a minimum, since the same functions are defined slightly different in the case of the flap being on the right. If we do not define them as functions here, we would have to include complicated *if*-evaluations later, which would add more lines of code and would have made our code much less readable. The functions for the flap being on the left look into the collection of points in the upper (or lower) quadrant and give back the index of the point which is mostly to the left. We also perform some actions to get that index number in the shape we prefer.

With the knowledge of the coordinates of the point that stands in for the tip for the top and bottom edge of the flap, we now want to know the coordinates for some points along the edge. We only have a collection of points in the top (and bottom) quadrant, but do not know how many and in what order they are stored. We want three pieces of information and it will be better to separate them out. These are the number of the point, its X coordinate, and its Y coordinate. The number is something we assign ourselves. A maximum number is set as four at the beginning and can be adjusted by hand. The *for*-loop takes care of this. Even though we might want four points along the edge, there may only be three, two, or just one. That is what the *if*-statement takes care of. We then find the index for the point with the lowest X value (if the flap is on the left) and use that index value to add the appropriate X and Y value to their respective lists. Then, we delete that point so that the next time *Get_upper_point_number* is called, it will not return the same point.

Once we have the X and Y coordinates of some points along the upper and lower edges of the flap, we can calculate the lengths of the triangle they make with the tip of the flap (or the point that stands in for the tip). This is done in a *for*-loop as it needs to be done separately for each point. Since the angle calculation depends on dividing by *A* (the length X-wise), and dividing by zero is not allowed, we incorporate a safety check and use a *try*-statement for the formula of the cosine law. Additionally, NumPy gives back an answer in radians, not degrees, for its *arccos*-function. So, we add a *rad2deg*-function to ensure the answer is provided in degrees. We end up with two lists: a couple of angles for the upper part of the flap, and a couple of angles for the lower part of the flap. In the variable *finalAngle*, we add up the averages of each and round it off to whole numbers and ensure we are left with only an integer, not a list. We have just programmed the computer to give us the angle of the flap!

7 Step 3: Running the Script over Large Numbers

In the script of this step, I have combined the previous two scripts into one and made it functionally different in the sense that we no longer want a visual representation of the angle, but want to have the computer analyze hundreds, possibly thousands, of manuscripts and output all of that data together. Instead of not giving the angle back, we actually give some information back, so that we can quickly see later how many manuscripts stopped working at which point in the code.

Most of the code is the same as before, with minor adjustments. We shall only discuss the important differences. At the beginning, we see an object defined with curly brackets, Python's way of saying this is a dictionary. Dictionaries are a lot like lists in the sense that you can stuff an arbitrary number of values in it. However, with the difference that everything you put in it, you must place them in pairs: the first value of the pair is the identifier, the keyword, the second word is the actual value. Thus, it works much like a real dictionary: for every keyword, there is a definition. These keywords must be unique, because if you wish to add something to the dictionary but use a previously used keyword, you do not add it but simply overwrite the value of the previous definition. Dictionaries are designed as though the order in which the entries are stored does not matter, and that is how they should be used. We will use a dictionary to save all the results and then write the dictionary to disk.

The function *Initiate_angle_finding* handles the whole process of analyzing an image, calling all other functions, and handling their responses. At the bottom of our code, we loop that function to cover all the manuscripts we want. We can either do so by looping through the number in which the filename ends, or simply by commanding the computer to inspect a certain folder and analyze all PDF files. This will generate a random order in which these files are analyzed and so it is useful to implement some *print* command to know the script is still running. This part of the script should be adapted if you have JPG files by uncommenting lines 71 and 72. If you work with a more complicated folder structure, you will have to find some regularity that you can exploit through a *for*-loop.

At the very end, a CSV file is created and saved on the hard disk. A CSV file is easy to import in, for instance, Excel or import back into Python. There is a *csv* package for Python which has specific functions to write all entries of a dictionary as a pair on a row.

```
# L.W. Cornelis van Lit, O.P. (c) 2017-2018
# Finding the angle of the flap of an Islamic manuscript

# ------------------------------------------------------------

# Dependencies
import cv2
import numpy as np
import io
from PyPDF2 import PdfFileMerger
import csv
import os
import time

#For analytical purpose only, a timer is set
start_time = time.time()

# A dictionary is created to catch the results
results = {}

# A string defining a JPG always starts and ends with these
values
startmark = b"\xff\xd8"
endmark = b"\xff\xd9"

#User variables
maximumEdgePoints = 4
kernelbig = np.ones((10, 10), np.uint8)

# ------------------------------------------------------------

def Initiate_angle_finding(pathOfFiles, nameOfPDF, pageOfPDF):

# Check if file exists and is sound. Change ".pdf" to .jpg"
to use jpgs instead of pdfs and uncomment lines 67 and 68.
  if not Accessing_file(pathOfFiles + nameOfPDF + ".pdf", 'rb'):
    results[nameOfPDF]="File inaccessible."

# Check if image on specific page can be extracted.
  else:
```

```
    extractedImage = Extract_image_from_PDF(pathOfFiles,
nameOfPDF, pageOfPDF)
    if not type(extractedImage) == np.ndarray:
      results[nameOfPDF] = "Image inaccessible."

# Perform analysis on image.
    else:
      hullContourX, hullContourY, imageWidth, imageHeight =
Analyze_image(extractedImage)
      if not hullContourX:
        results[nameOfPDF] = "Cannot analyze image."
      else:
        angleInfo = Find_angle(hullContourX, hullContourY,
imageWidth, imageHeight)
        results[nameOfPDF] = angleInfo

# ------------------------------------------------------------

# Check if a file exists and is accessible.
def Accessing_file(filepath, mode):
  try:
    f = open(filepath, mode)
    f.close()
  except:
    #File unreadable.
    return False
  #File readable.
  return True

# ------------------------------------------------------------
# This function takes one PDF file and extracts one page from
it, to save as JPG.
# Only to be used on PDFs in which each page is only an image.

def Extract_image_from_PDF(pathOfFiles, nameOfPDF, pageOfPDF):

#Uncomment these two lines of code if files are already images:
# img = cv2.imread(pathOfFiles + nameOfPDF + ".jpg")
# return img
```

```
# File exists, so prepare the virtual containers.
  merger = PdfFileMerger()
  virtualpdf = io.BytesIO()

# Opens PDF and takes out specific page
  with open( pathOfFiles + nameOfPDF + ".pdf", "rb") as
sourcePDF:
    try:
      merger.append(fileobj=sourcePDF, pages=(pageOfPDF - 1,
pageOfPDF))
      merger.write(virtualpdf)
    except:
      return
  merger.close()

# Read the desired page simply in its string of values
  pdf = virtualpdf.getvalue()
  virtualpdf.close()

# Find the start and end of the string defining the JPG
  jpgstart = pdf.find(startmark, 0)
  jpgend = pdf.find(endmark, jpgstart)

# Reading out the entire string defining the JPG
  jpgstring = pdf[jpgstart:jpgend]

# Preparing the string to be read by OpenCV
  jpgstring2 = np.fromstring(jpgstring, np.uint8)

# Turn into OpenCV-readable image
  image = cv2.imdecode(jpgstring2, 1)

# Give back OpenCV-readable image
  return image

# ------------------------------------------------------------------

#This function analyses the image.
def Analyze_image(image):
  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
  ret, thresh1 = cv2.threshold(gray, 180, 255, cv2.
THRESH_BINARY_INV)
  # Twice scaling is part of accessing the angle
  scaled = cv2.resize(thresh1, (0, 0), fx=0.8, fy=0.8)
  openimg = cv2.morphologyEx(scaled, cv2.MORPH_OPEN, kernelbig)
  ret, thresh2 = cv2.threshold(openimg, 1, 255, cv2.
THRESH_BINARY)
  scaledagain = cv2.resize(thresh2, (0, 0), fx=0.5, fy=0.5)

  imageHeight, imageWidth = scaledagain.shape

  _, contours, _ = cv2.findContours(scaledagain, cv2.RETR_
EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
  try:
    mainContour = max(contours, key=len)
  except:
    return (None,None,None,None)

  hull = cv2.convexHull(mainContour, returnPoints=False)

  hullContourX = []
  hullContourY = []
  for correctYvalue in hull:
    hullContourX.append(mainContour[correctYvalue[0]][0][0])
    hullContourY.append(mainContour[correctYvalue[0]][0][1])

  mainContourX = []
  mainContourY = []

  mainContourTotalX = 0
  mainContourTotalY = 0
  for correctYvalue in range(0,len(mainContour)):
    mainContourX.append(mainContour[correctYvalue][0][0])
    mainContourY.append(mainContour[correctYvalue][0][1])
    mainContourTotalX = mainContourTotalX +
mainContour[correctYvalue][0][0]
    mainContourTotalY = mainContourTotalY +
mainContour[correctYvalue][0][1]

  return hullContourX, hullContourY, imageWidth, imageHeight
```

```
# ----------------------------------------------------------------

def Find_angle(hullContourX, hullContourY, imageWidth,
imageHeight):
  # Using NumPy requires arrays
  reducedXArray = np.array(hullContourX)
  reducedYArray = np.array(hullContourY)

  # Using NumPy to get index numbers of all Y values within
15% of middle
  reducedYLength = len(hullContourY)
  varianceY = imageHeight * 0.15
  minimumY = imageHeight / 2 - varianceY
  maximumY = imageHeight / 2 + varianceY
  decideMiddlePoints = np.zeros(reducedYLength, dtype=np.int)

  for correctYvalue in range(0, reducedYLength):
    if minimumY <= hullContourY[correctYvalue] <= maximumY:
      np.put(decideMiddlePoints, correctYvalue, 1)
      xVal = hullContourX[correctYvalue]

      # Ruling out spurious points on non-flap side by
checking if nearby points in terms of x-value are near edge
in terms of y-value (i.e. in a corner)
      # Also ruling out flaps with distortions, i.e. deleting
all middle points on the left/right side of the image where
there is a point which meets above requirement.
      for otherXvalues in range(0, reducedYLength):
        if (xVal - 5 <= hullContourX[otherXvalues] <= xVal
+ 5) and (hullContourY[otherXvalues] < imageHeight * 0.2 or
hullContourY[otherXvalues] > imageHeight * 0.8):
          if hullContourX[otherXvalues] <= imageWidth/2:
            np.put(decideMiddlePoints,np.where( reducedXArray
< imageWidth / 2), 0)
          else:
            np.put(decideMiddlePoints, np.where(reducedXArray
> imageWidth / 2), 0)
          break
```

```
  # Only proceed if a flap can be found.
  # Get X,Y value of tip-point, making sure there are some
points that could be the tip

  if not np.count_nonzero(decideMiddlePoints) == 0:
    # Get X,Y value of only tip-points
    tipOfFlapYArray = reducedYArray[decideMiddlePoints != 0]
    tipOfFlapXArray = reducedXArray[decideMiddlePoints != 0]
    restOfHullXArray = reducedXArray[decideMiddlePoints == 0]
    restOfHullYArray = reducedYArray[decideMiddlePoints == 0]

    # tipX needs to be decided, left or right
    # Is the tip on the left?
    if np.average(tipOfFlapXArray) < imageWidth / 2:
      # Get X,Y value of furthest tip-point
      reducedXMinimumArray = np.where(tipOfFlapXArray ==
tipOfFlapXArray.min())
      reducedXMinimumList = reducedXMinimumArray[0].tolist()
      tipX = int(tipOfFlapXArray.min())

    # Is the tip on the right?
    elif np.average(tipOfFlapXArray) > imageWidth / 2:
      reducedXMinimumArray = np.where(tipOfFlapXArray ==
tipOfFlapXArray.max())
      reducedXMinimumList = reducedXMinimumArray[0].tolist()
      tipX = int(tipOfFlapXArray.max())

    else:
      return ("Middle points undetermined.")
  else:
    if len(hullContourX) > 4:
      return ("Cannot find flap.")
    else:
      return ("No flap.")

  # Check if hull is formed correctly
  if (abs(restOfHullXArray - tipX) < 4).any():
    return ("Some point of hull too close to X value of tip.
Hull incorrectly identified.")
```

```
  # Now find points along the edge and calculate angle

  tipUpperX = tipX
  tipUpperY = np.min(tipOfFlapYArray[reducedXMinimumList])
  tipLowerX = tipX
  tipLowerY = np.max(tipOfFlapYArray[reducedXMinimumList])

  #For left flap we need to look at minimal X values, for
right flap maximal X values.
  if tipX < imageWidth / 2:

    # Get all XY that are in the upper quadrant
    upperRestOfHullXArray = restOfHullXArray[restOfHullYArray
< imageHeight / 2]
    upperRestOfHullYArray = restOfHullYArray[restOfHullYArray
< imageHeight / 2]

    upperRestOfHullYArray = upperRestOfHullYArray[upperRestOf
HullXArray < imageWidth * 0.2]
    upperRestOfHullXArray = upperRestOfHullXArray[upperRestOf
HullXArray < imageWidth * 0.2]

    # Get all XY that are in the lower quadrant
    lowerRestOfHullXArray = restOfHullXArray[restOfHullYArray
> imageHeight / 2]
    lowerRestOfHullYArray = restOfHullYArray[restOfHullYArray
> imageHeight / 2]

    lowerRestOfHullYArray = lowerRestOfHullYArray[lowerRestOf
HullXArray < imageWidth * 0.2]
    lowerRestOfHullXArray = lowerRestOfHullXArray[lowerRestOf
HullXArray < imageWidth * 0.2]

    def GetUpperPointNumber():
      return((np.where(upperRestOfHullXArray ==
upperRestOfHullXArray.min())[0]).tolist())

    def GetLowerPointNumber():
```

```
      return((np.where(lowerRestOfHullXArray ==
lowerRestOfHullXArray.min())[0]).tolist())
  else:
    # Get all XY that are in the upper quadrant
    upperRestOfHullXArray = restOfHullXArray[restOfHullYArray <
imageHeight / 2]
    upperRestOfHullYArray = restOfHullYArray[restOfHullYArray
< imageHeight / 2]

    upperRestOfHullYArray = upperRestOfHullYArray[upperRestOf
HullXArray > imageWidth * 0.8]
    upperRestOfHullXArray = upperRestOfHullXArray[upperRestOf
HullXArray > imageWidth * 0.8]

    # Get all XY that are in the lower quadrant
    lowerRestOfHullXArray = restOfHullXArray[restOfHullYArray >
imageHeight / 2]
    lowerRestOfHullYArray = restOfHullYArray[restOfHullYArray >
imageHeight / 2]

    lowerRestOfHullYArray = lowerRestOfHullYArray[lowerRestOf
HullXArray > imageWidth * 0.8]
    lowerRestOfHullXArray = lowerRestOfHullXArray[lowerRestOf
HullXArray > imageWidth * 0.8]

    def GetUpperPointNumber():
      return((np.where(upperRestOfHullXArray ==
upperRestOfHullXArray.max())[0]).tolist())
    def GetLowerPointNumber():
      return((np.where(lowerRestOfHullXArray ==
lowerRestOfHullXArray.max())[0]).tolist())

  # Establishing upper edge points
  upperEdgePoint = []
  upperEdgeX = []
  upperEdgeY = []
  for pointNumber in range(0, maximumEdgePoints):
    if pointNumber < len(upperRestOfHullXArray):
```

```
        upperEdgePoint.append( GetUpperPointNumber() )
        upperEdgeX.append( upperRestOfHullXArray[upperEdgePoint
[pointNumber][0]] )
        upperEdgeY.append(upperRestOfHullYArray[upperEdgePoint[
pointNumber][0]])
        upperRestOfHullXArray = np.delete(upperRestOfHullXArray,
upperEdgePoint[pointNumber])
        upperRestOfHullYArray = np.delete(upperRestOfHullYArray,
upperEdgePoint[pointNumber])

  # Establishing lower edge points
  lowerEdgePoint = []
  lowerEdgeX = []
  lowerEdgeY = []
  for pointNumber in range(0, maximumEdgePoints):
    if pointNumber < len(lowerRestOfHullXArray):
        lowerEdgePoint.append( GetLowerPointNumber() )
        lowerEdgeX.append(lowerRestOfHullXArray[lowerEdgePoint[
pointNumber][0]])
        lowerEdgeY.append(lowerRestOfHullYArray[lowerEdgePoint[
pointNumber][0]])
        lowerRestOfHullXArray = np.delete(lowerRestOfHullXArr
ay, lowerEdgePoint[pointNumber])
        lowerRestOfHullYArray = np.delete(lowerRestOfHullYArr
ay, lowerEdgePoint[pointNumber])

  # !!! Note that upper/lowerRestOfHull are now empty !!!

  # Calculate degree of angle based on lower edge points
  # First apply Theorem of Pythogaras to find all lengths,
with A = horizontal, B = vertical, C = diagonal

  upperAngles = []
  for pointNumber in range(0,len(upperEdgeX)):
    # First apply Theorem of Pythogaras to find all lengths,
with A = horizontal, B = vertical, C = diagonal
    A = abs(tipUpperX - upperEdgeX[pointNumber])
    B = abs(tipUpperY - upperEdgeY[pointNumber])
    C = A**2 + B**2
```

```
    # Then use Law of Cosine to find angle. Answer is returned
in Radians so it needs to be transposed to Degrees for human
readability.
    try:
      upperAngles.append( np.rad2deg(np.arccos((A**2 + C -
B**2) / (2 * A * np.sqrt(C)))) )
    except:
      return ("Upper points incorrectly identified.")

  lowerAngles = []
  for pointNumber in range(0, len(lowerEdgeX)):
    A = abs(tipLowerX - lowerEdgeX[pointNumber])
    #A is 0 which is causing the problem. Not allowed to divide
by 0.
    B = abs(tipLowerY - lowerEdgeY[pointNumber])
    C = A**2 + B**2
    try:
      lowerAngles.append(np.rad2deg(np.arccos((A ** 2 + C - B
** 2) / (2 * A * np.sqrt(C)))))
    except:
      return ("Upper points incorrectly identified.")

  # Entire angle, rounded to zero decimals (would give false
sense of accuracy otherwise) is:
  if not upperAngles or not lowerAngles:
    return ("Cannot make out angle.")
  else:
    finalAngle = int(np.rint( np.average(np.
asarray(lowerAngles)) + np.average(np.asarray(upperAngles)) ))
    return(finalAngle)

# --------------------------------------------------------------

collectionName = "Nuruosmaniye-1-500"
directoryPath = "/Volumes/ManuscriptsHD/" + collectionName +"/"
directory = os.fsencode(directoryPath)

for file in os.listdir(directory):
  filename = os.fsdecode(file)
```

```
  if filename.endswith(".pdf"):
    if not filename.endswith("_text.pdf"):
      filename = filename[:-4]
      print("Working on manuscript " + filename)
      Initiate_angle_finding(directoryPath,filename,1)

#We can also loop the function over the number in which the
filename ends by using these lines of code:
#collectionName = "NURUOSMANIYE"
#startNumber = 1
#finishNumber = 362
#for i in range(startNumber,finishNumber+1):
# print("Working on manuscript " + collectionName + " " +
str(i))
# Initiate_angle_finding("/Volumes/ManuscriptsHD/Nuruosmaniye/"
,collectionName+str(i), 3)

# Save results as CSV
with open(collectionName+".csv", "w") as output:
  writeToCSV = csv.writer(output)
  writeToCSV.writerows(results.items())

print("Finished in %s seconds." % (time.time() - start_time))
```

8 Results

Arguably, what we created so far is not a final research result in its own right but only a stepping stone. More scripts could be developed to analyze the data we generated; for example, we could clean our data by doing our best to get rid of false positives. This could be achieved with a script to display the lowest and highest results to visually inspect whether they have correctly been calculated or whether the computer could not deduce the angle correctly. This could simply include a user interface where, with the keys Y and N, we could accept or decline the results; and if we decline, the result would be deleted from the CSV file. There are other issues that might muddy our data. For example, maybe we want to delete those codices which are not in their original binding, and so we would need to find further regularities to exploit. One possibility is to analyze the entire shape of a codex and compare it to the shapes of all the other codices, and if there is a (near)perfect match, we could flag it as a possible rebind.

Perhaps our input comes from digitized microfilms, and we might need to alter the algorithms to analyze those particular kinds of images.

After data cleaning, we could develop scripts to dynamically load all results and visualize in graphs the statistics that they produce, for instance, by using the *MatPlotLib* package. Even better, we could try to obtain other metadata on the manuscripts, for instance their date and origin, to try to relate the angle to specific eras and places. On the other hand, we could see the angle detection as only a first step to detect other features of the manuscript that we might be able to pinpoint if we can verify that certain points belong to the tip and the edges of the flap. Here is not the place to go into those things. Let us instead go over the final results.

TABLE 7.1 Number of manuscripts per outcome

Hull incorrectly identified	39
No flap	214
Image inaccessible	38
File inaccessible	83
Cannot make out angle	10
Cannot find flap	532
Angles	1,084
Total	**2,000**

In total, I processed approximately two-thousand manuscripts, which took me about half a day. The number of angles calculated seems disappointingly low given the total. However, this was expected as many of the files that I had access to did not have a photo of the cover of the codex; and for some that did, they did not have that photo on the first page. A logical step would be to look into the 'cannot find flap' category and see if it consists mostly of false negatives. If so, perhaps the code could be adjusted to collect more angles. I did not attempt this at this point. Checking parts of the processed images by hand yielded a success rate of 98%. The two percent false positives had to do with extra objects remaining in the background which OpenCV analyzed as part of the codex.

Of the manuscripts for which the angle could be determined, it became clear that there is a pretty significant concentration around certain angles. The number of codices that were identified as having a flap with an angle below 150° was 8, accounting for 0.7% of the total. Codices with a flap above 170° were 7 in total, or 0.6%. That means that 98.7% of the codices has a flap with

an angle between 150° and 170°. Further, degrees 156 through 160 are the only ones with a count above a hundred, and 158° sticks out high above the rest, with 165 manuscripts. The diagram at the end of this chapter shows the number of manuscripts with a flap that has an angle for a certain degree. The angle can obviously only be between 0° and 180°. Among the ones that are counted below 150° and above 170°, we should assume there are some false positives. Nevertheless, I do not think we need to clean our data as the vast number of manuscripts convincingly indicate the matter here, namely that the flap (*lisān*) of an Islamic manuscript will normally make an angle of about 156° to 160° degrees, as this narrow bandwidth already accounts for 60% of all manuscripts. That being said, clearly the flap of codices was not produced with a standardized tool, as the other 40% falls outside of this bandwidth and we do see considerable variety.

With this chapter, then, I have shown that codicology too can benefit from the mass digitization of manuscripts. Python and OpenCV are powerful and stable pieces of technology. Even if they will be superseded, the skills gained from using them will be easily transferrable to other programming languages. What is most important is not to know certain commands and functions by heart, but to have a general idea what a programming language or a package is capable of, to write out pseudo-code along those lines, and then to find out how to exactly implement this from a technical point of view. The technology might still throw you some curve balls. But workarounds often present themselves in online discussions among people who have previously dealt with a similar issue. As long as your code achieves what you want, is readable, and reasonably flexible, you do not need to worry about the rest. As a result, I maintain that virtually anybody can instruct a computer to perform fairly advanced analyses on images of manuscripts, automated over large amounts of data.

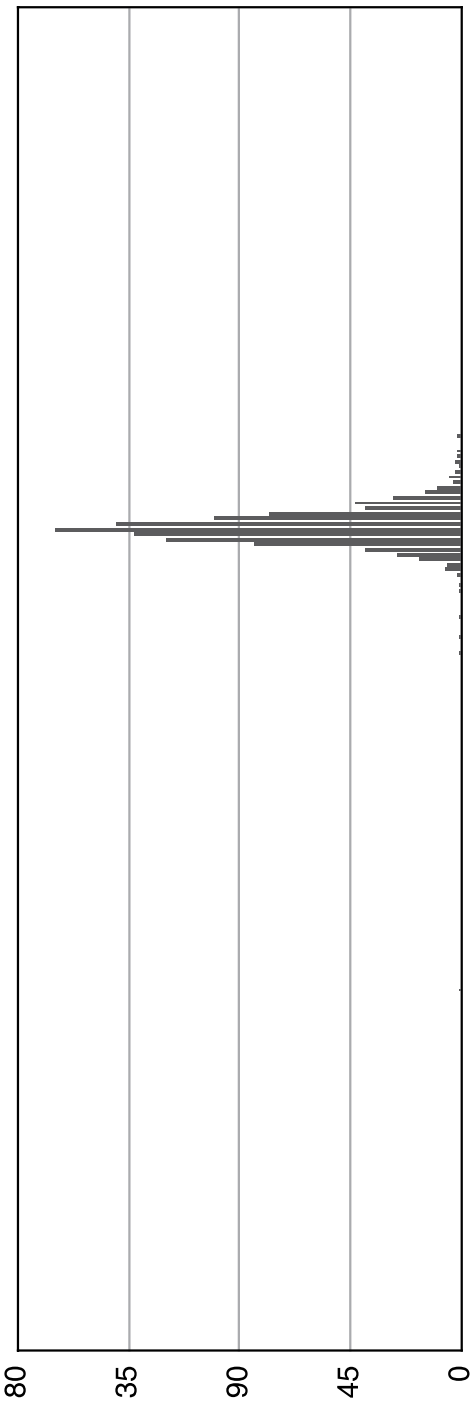

FIGURE 7.2 Bar chart of measured angles for a thousand manuscripts

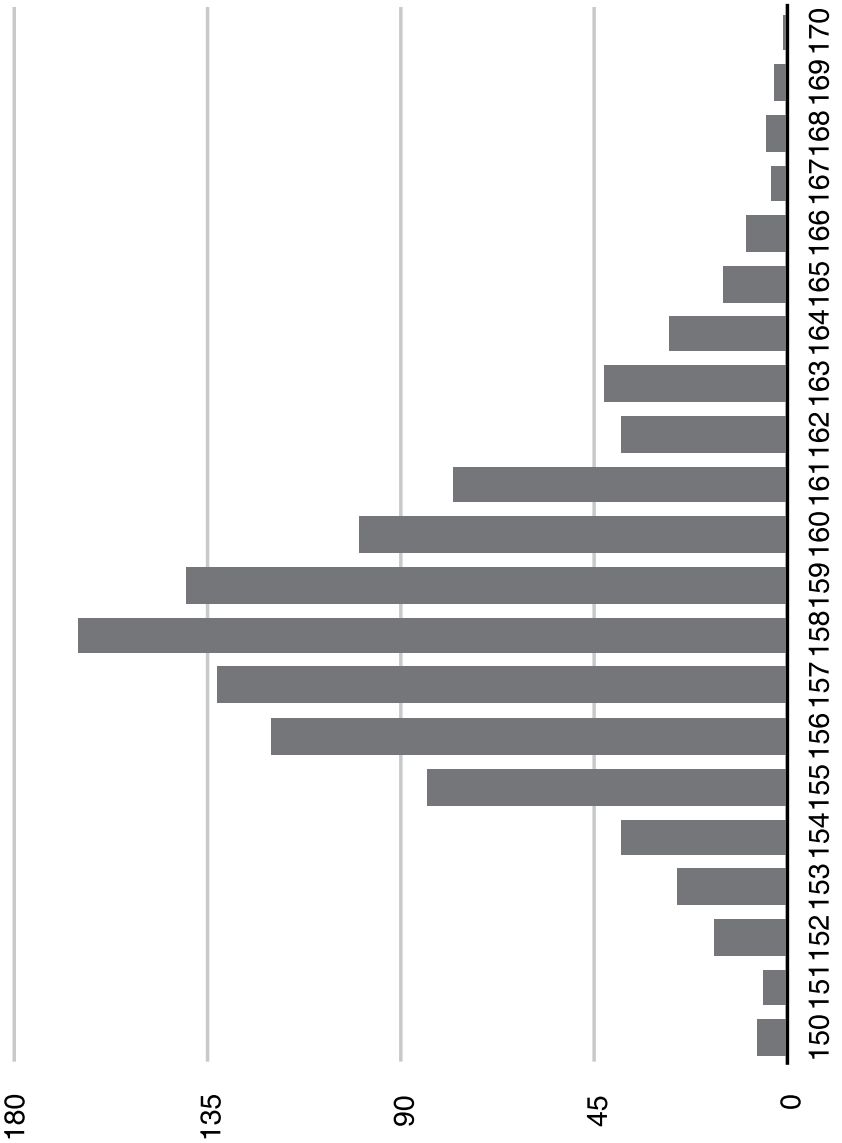

FIGURE 7.3 Close up showing most manuscripts have a flap with an angle of around 158°